NASA CR-159,301

NASA Contractor Report 159301

NASA-CR-159301
19800020547

FUNCTIONAL SPECIFICATION OF THE PERFORMANCE MEASUREMENT (PM) MODULE

Jeffrey E. Berliner

BOLT BERANEK AND NEWMAN INC.
Cambridge, Massachusetts 02138

NASA Contract NAS1-15192
July 1980

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Table of Contents

N80-29048#

List of Figures

1. Executive Summary

1.1 Introduction

The objective of the Performance Measurement (PM) Module is to provide a computer-based interactive system for collecting and analyzing data from a wide variety of experiments.

The challenge in the design of the PM Module is to provide a system that will be broad and flexible enough to handle a wide variety of experiments and performance measures, some of which have not yet been conceived.

This document describes the design of the Performance Measurement Module. It focuses primarily on what the PM Module would do, and what it would look like to the user. The PM Module as described here could take several man-years to develop. This report suggests an evolutionary approach to the implementation of the PM Module. With such an approach an operational baseline PM Module could be running within a few months.

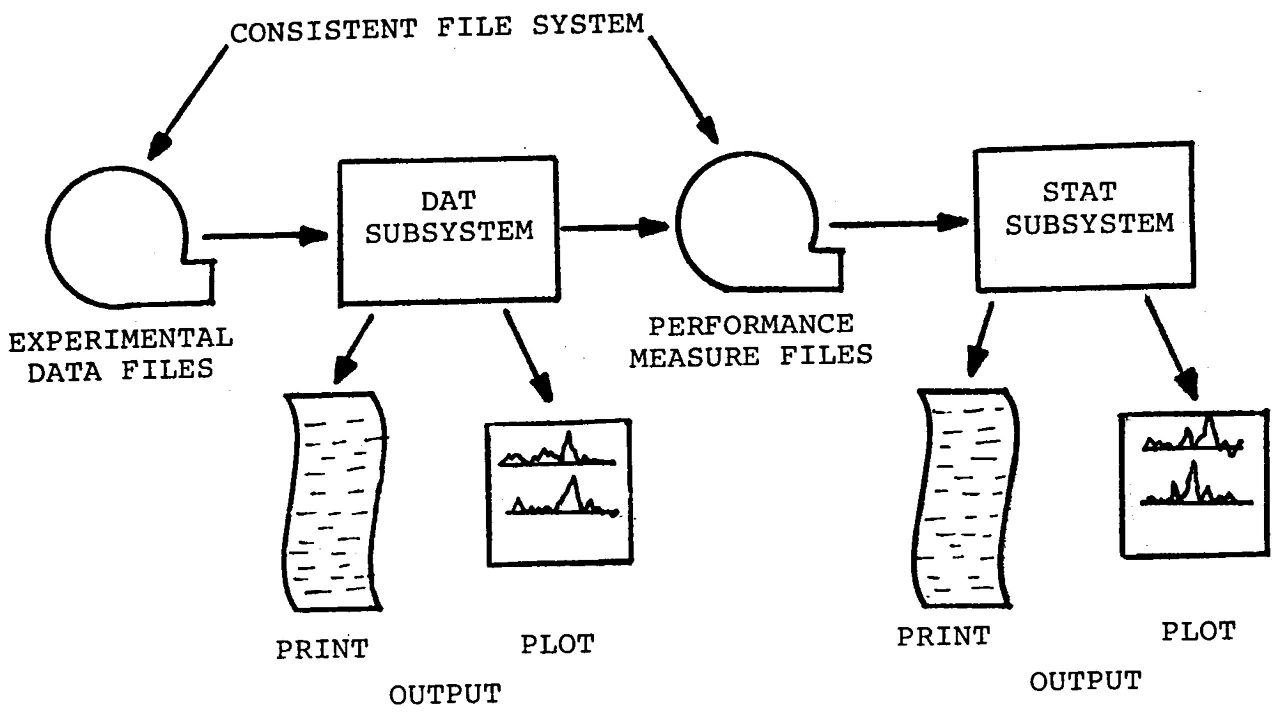
1.2 Overview of the PM Module: DAT/STAT

The Performance Measurement Module consists of two subsystems: a Data Analysis and Transformation (DAT) subsystem and a Statistical Analysis (STAT) subsystem. These subsystems are linked by a Consistent File System. See Figure 1.1. Files containing raw experimental data, such as time histories, are read and processed using the DAT subsystem, producing files of performance measures. These files, in turn, would be read and processed by the STAT subsystem.

The DAT subsystem contains printing and plotting commands for examining raw data and derived performance measures. The STAT subsystem contains printing and plotting commands for examining the performance measures and the statistical summaries.

There are primarily two reasons for dividing the PM Module into these two subsystems. First, the primary functions of these two subsystems are quite distinct: data manipulation versus statistical testing, although their secondary functions may indeed overlap. Second, it appears likely that it will be possible to purchase a statistical analysis package which would serve quite well as the STAT subsystem.

Overviews of the DAT and STAT subsystems are presented below in Sections 1.2.1 and 1.2.2., and an overview of the Consistent File System is presented in Section 1.2.3. A plan for the evolutionary development of the PM Module is presented in Section 1.3. The DAT



DAT: DATA ANALYSIS & TRANSFORMATION

STAT: STATISTICAL ANALYSIS

Figure 1.1 Functional Block Diagram of the PM Module

subsystem and programming language are described in Sections 2 and 3, respectively. The STAT subsystem is described in Section 4, and the Consistent File System for the PM Module is described in Section 5.

1.2.1 Overview of the DAT Subsystem

The purpose of the Data Analysis and Transformation (DAT) subsystem is to provide a tool for the scientific investigator to examine the data from his experiments. A useful analogy for this subsystem is a programmable matrix calculator with a graphic display. The current form of this subsystem has evolved over the past several months, and is now based strongly on BBN's RS/1 data management system. The subsystem began as a unified collection of relatively fixed analysis programs, and has evolved to the current more integrated and flexible form.

The DAT subsystem, which is operated using a simple English-like command language, assists the user in managing two-dimensional data tables, in visualizing this data through graphs, and in preparing the data for statistical analysis via the Statistical Analysis (STAT) subsystem. In addition, the DAT subsystem includes a programming language which permits the creation of new procedures. There are two major purposes for these user-defined procedures. First, they enable the user to automate a sequence of analysis steps, such as reading a file, computing some measures, and plotting the results. Second, using the DAT programming language, the user can conveniently create and modify new performance measures and try them out on actual data.

The functional description of the DAT subsystem is divided into two parts: a description of the DAT commands and a description of the DAT programming language.

1.2.2 Overview of the STAT Subsystem

The purpose of the Statistical Analysis (STAT) subsystem is to provide a tool for the scientific investigator to perform statistical tests and make statistical inferences from his data.

As part of the design of the PM Module, a survey was made of existing, commercially available statistical analysis packages. It was hoped that such a package could be incorporated into the PM Module, thereby avoiding needless extensive software development.

In order to be considered for use as the STAT subsystem, a statistical package would have to meet the following four basic requirements:

(1) Includes a large range of **standard statistical analyses**, including analysis of variance.

(2) Runs **interactively**.

(3) Runs on a **CDC Cyber-175** computer.

(4) Provides for **data transformation**.

The fourth requirement is the loosest. With a powerful data transformation capability, a statistical analysis package could fulfill virtually all the requirements of the entire PM Module. In fact, we suggest in the next section that the PM Module be developed in an evolutionary manner, with the first step being an Baseline PM Module in which the DAT subsystem does little more than read and write files, and the data transformation capabilities of the STAT subsystem suffice.

Currently, there are two candidate statistical analysis packages under consideration:

(1) **P-STAT 78** produced by P-STAT, Inc. of Princeton, New Jersey.

(2) **SIPS** produced by the Department of Statistics of Oregon State University, Corvallis, Oregon.

A number of other packages were also considered, but have been eliminated for one reason or another. In Section 4 of this document, we present the requirements of the STAT subsystem in more detail, we compare the various candidate packages, and make a recommendation. The final decision on selecting a statistical package is left to the LRC staff, and will be based in part on experience with the packages gained in a trial period which is just now beginning.

1.2.3 Overview of the Consistent File System

The DAT and STAT subsystems are linked by a Consistent File System. This file system provides a means for passing data from the output of the experiments through the DAT and STAT subsystems. The primary task of the Consistent File System is to maintain the correspondence between the value of the data (e.g. the value of a particular performance measure) and the identities of the data (e.g. the name of the performance measure).

The design of the Consistent File System for the PM Module has two distinct parts: the design of **internal file structures** and the design of **external file structures**. Internal file structures means the choice of header content, record size, etc.: how does one identify or find a particular datum within a file. External file structures means the choice of file naming and accessing conventions: how does one identify and retrieve a particular file for analysis.

The Consistent File System, described in Section 5, includes internal file structures which are designed meet the following objectives:

(1) **Adaptable**, so that a wide variety of experiments and analyses may be accommodated.

(2) **Self-documenting**, so that the files may be shared by users with little contact with each other.

(3) **Expandable**, so that a subset of the users can make an addition to the file system.

(4) **Upward compatible**, so that features which are not incorporated during the initial development can be added later.

(5) **Backward compatible**, so that existing file handling software (e.g. SIFT) can be used.

(6) **Convenient**, so that reading and writing the files requires just a few lines of code.

(7) **Efficient**, so that the computing resources are not unduly burdened.

The Consistent File System also includes external file structures which simplify the task of identifying and retrieving a particular file for analysis. The system is ,in fact, a file-naming convention, supported by appropriate software, with multi-field file names. The fields would serve to identify the following attributes of the file:

(1) The **user** who "owns" the file: his group and his name.

(2) The **experiment** to which the file pertains: the name of the experiment, the subject and run number.

(3) The **type** of file: time-series data, processed data, etc.

(4) The **version** number of the file.

These file naming conventions would probably also be very useful for keeping track of programs pertaining to the PM Module, such as FORTRAN source files, documentation files, etc.

1.3 Evolutionary Development of the PM Module

The preceding section (1.2) was a functional description of a rather complete PM Module. Such a module could take several man-years to develop. This section describes an evolutionary approach to the implementation of the PM Module. In this approach, the ultimate design would remain as described above. The approach to this goal, however, would be via an evolutionary development, beginning with a baseline PM Module of somewhat limited power, and proceding via a series of upgrades to a complete PM Module.

There are two very important advantages to this evolutionary development. First, it should be possible to have an operational baseline PM Module running within a few months. Second, it allows the experience gained in utilizing the early stages of the PM Module to guide the design of the later stages.

In the next two sections, an overview of the baseline PM Module is given, followed by a tentative series of upgrades.

1.3.1 Overview of the Baseline PM Module

The purpose of the Baseline PM Module is to provide a useful, working system with a minimum development effort. The form of the Baseline PM Module is shown in Figure 1.2. It consists of the statistical analysis package selected for the STAT subsystem, with some additional code so that the consistent file system could be utilized to read time history data files and to write performance measurement files.

The DAT subsystem implemented in this Baseline PM Module would be rather primitive. It would include a workable subset of the functions of the DAT subsystem. It would be implemented by adding a few new functions (such as FFT and plot functions) to the data transformation capabilities of the STAT subsystem.

1.3.2 Evolutionary Upgrades to the Baseline PM Module

The Baseline PM Module, as described in the previous section, falls short of the full PM Module in four major areas:

(1) interactive graphics,

(2) interactive data transformation,

(3) automated data transformation, and

(4) improved conversational front-end.

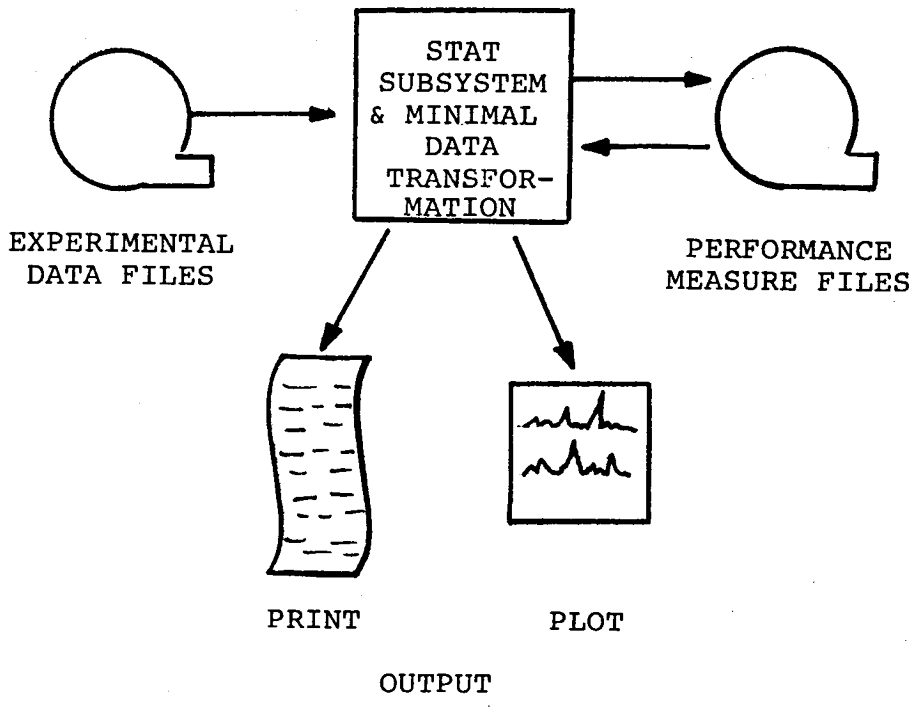

STAT SUBSYSTEM & MINIMAL DATA TRANSFORMATION
EXPERIMENTAL DATA FILES
PERFORMANCE MEASURE FILES
PRINT
PLOT
OUTPUT

Figure 1.2 Form of the Baseline PM Module

Once the Baseline PM Module is running, these three areas could be upgraded to the level of the full PM Module.

The first area to be upgraded would almost certainly be interactive graphics. It is widely recognized that graphical presentation of numerical results can be of tremendous value to the person exploring a set of experimental results. The Baseline PM Module provides a very limited capability in this area, it would be an appropriate place to begin the upgrade.

The next area to be upgraded might be the data transformation capabilities. Whether the work began with interactive data transformation (i.e. building towards the DAT subsystem command language), or automated data transformation (i.e. building towards the DAT programming language), or both together, would depend on the experience gained with the early utilization of the Baseline PM module.

The decision to add a more conversational front-end to the PM Module would depend on the type of people who were expected to use the system. Such a front-end could be very attractive to the scientist who is relatively unfamiliar with computer-based systems. On the other hand, other users might find that it introduced an unnecessary overhead.

2. The DAT Subsystem

2.1 DAT Subsystem Concepts

The DAT subsystem provides a convenient and efficient way to collect, store, analyze and examine research data. The user provides the system with numeric or textual data in tabular form from either an existing file, or from the computer terminal keyboard. Using a simple command language, the user can revise the data tables, create new data tables, display the data tables, create graphs of the data, revise the graphs, display the graphs on the terminal, and save the tables and graphs as files for later use.

Although the DAT subsystem's command language provides a powerful set of tools for data manipulation and analysis, a user may create new procedures using the DAT programming language. This simple programming language allows the user to perform all of the DAT subsystem's commands under program control. The two major purposes for these user-defined procedures is (1) to automate frequently performed sequences of analysis steps, and (2) to conveniently create and modify new performance measures.

The DAT command language is described in this section (2.), and the DAT programming language is described in the next section (3.).

Although the DAT subsystem has little or no statistical analysis capabilities built into it, it is easy for the user to write data tables as files which would be read by the STAT subsystem.

2.1.1 A Very Short Session

The DAT subsystem is a powerful system which includes table structures for holding data, making graphs, etc. However, in this introductory session, the DAT subsystem will be used as a simple desk calculator. This session will introduce a number of concepts which will be explained later.

When the DAT subsystem is ready to accept a command, it types the prompt character "#". From that point on, until the user types the GO character (ALT MODE or ESCAPE), the system is waiting for the user to finish typing. When the GO character is typed, it begins to execute the command. It is not ready to accept another command until "#" is typed again.

The following session uses only the TYPE and SET commands:

```
#TYPE 3.14159 <GO>
3.14159
```

```
#TYPE PI <GO>
EMPTY
#SET PI TO 3.14159 <GO>
#TYPE PI <GO>
3.14159
#TYPE PI*10 <GO>
31.4159
#TYPE 'DIAMETER OF CIRCLE IS: ', 10 <GO>
DIAMETER OF CIRCLE IS:
10
#SET RADIUS TO 5.7 <GO>
#TYPE NOCR 'RADIUS OF CIRCLE IS:', RADIUS,'
DIAMETER OF CIRCLE IS:', 2*RADIUS,'
AREA OF CIRCLE IS:', PI*RADIUS**2 <GO>
RADIUS OF CIRCLE IS: 5.7
DIAMETER OF CIRCLE IS: 11.4
AREA OF CIRCLE IS: 102.07
#QUIT <GO>
```

2.1.2 Common DAT Commands: TYPE and SET

The DAT subsystem manipulates and displays data by executing commands which the user types in to it. A common command, which is used in the above very short session, is the TYPE command, which tells the DAT subsystem to type a value. The user can specify several values to TYPE by separating the values by commas. Each value will be typed on a new line, unless the TYPE NOCR command is used. "NOCR" stands for no carriage return. The general form for the TYPE statement is:

TYPE [NOCR] {value}

where [...] indicates an optional argument, and {...} indicates that several arguments may be present separated by commas.

The SET command is used to change the value of a variable; it does not cause anything to be typed out. The general form of the SET command is:

SET variable TO expression

A useful, although not necessary, feature of the DAT subsystem would be to allow the use of "=" interchangeably with the set command:

variable = expression

Variables are used to store results that may be used later. The user need not do anything special to make a variable exist; the DAT subsystem takes care of that when a variable is used for the first time. It also has an explicit representation for data which is not known, the value EMPTY. This is the value all variables have when they are first created.

2.1.3 Values

There are four different kinds of data values in the DAT subsystem:

FIXED - An integer, e.g. 4, -5, 0, 15123

FLOAT - A number with a decimal point, or written with an "E" followed by a number, to mean "times 10 to the number." For example, 3.14159, 3.5E12, 2E-7

TEXT - Text values are strings of typed characters, enclosed in single quotes. For example, ´THIS IS A TEXT.´, ´3´, ´RADIUS OF CIRCLE IS:´

EMPTY - This is a special value. It is used to indicate that the value is not known. EMPTY may be used in computations, but it typically produces an EMPTY result. For example, EMPTY + 1 is EMPTY.

2.1.4 Expressions: Operators and Functions

Values may be computed in the DAT subsystem by means of expressions. An expression is formed by using operators and functions to combine constants and variables.

The order of evaluation of an arithmetic expression may be explicitly indicated by parentheses. Where this is not done, expressions are evaluated in the following order:

(1) **(exponentiation) and - (unary minus)

(2) * and / (multiplication and division)

(3) + and - (addition and subtraction)

Within these constraints, expressions are evaluated from left to right. Thus A/B*C is evaluated as (A/B)*C. The only exception to this left-to-right rule is exponentiation where A**B**C is evaluated as A**(B**C).

There are four categories of operators available in the DAT subsystem:

Arithmetic: +, -, *, /, **

Relational: EQ, NE, LT, GT, LE, GE

Logical: AND, OR, NOT

Text: CAT (concatenation)

There are six categories of functions available in the DAT subsystem:

Trigonometric: SIN, COS, TAN, ...

Numerical: SQRT, EXP, LOG, MAX, MIN, ...

Text: CHARS, CAT, ...

Data Type: TNUM, TYPE, EMPTY, NUM.TO.TEXT, ...

Graphic: DLINE, DMOVE, ERASE, ...

Miscellaneous: GETFILE, PUTFILE, YESANSWER, ...

2.2 DAT Tables

One of the most important entities in the DAT subsystem is the data table. Figure 2.1 is a typical table. It contains data from a realistic (though fanciful) data set concerning the influence of a drug called TAIL-GRO on the growth of armadillo tails.

2.2.1 Making a Table via the Keyboard

This table was constructed using the MAKE TABLE command. This command prompts the user for the title of the table, row names and column headings, and for the data values comprising the body of the table.

The MAKE TABLE dialogue that led to the table in Figure 2.1 is given below:

```
#MAKE TABLE ARMA <GO>

TITLE OF TABLE? ARMADILLO TAIL-GRO DATA <GO>
ARE THERE ROW NAMES? YES <GO>
ARE THERE COLUMN HEADINGS? YES <GO>
ENTER COLUMNWISE? YES <GO>
NAME OF ROW 1 = A352C <GO>
NAME OF ROW 2 = A601D <GO>
NAME OF ROW 3 = A705R <GO>
NAME OF ROW 4 = A852R <GO>
NAME OF ROW 5 = A965D <GO>
```

ARMA 3C X 7R ARMADILLO TAIL-GRO DATA

	1 PERSONALITY	2 DOSE (MG)	3 TAIL (CM)
1. A352C	FRIENDLY	8.2	7.9
2. A601D	QUIET	13.2	24.3
3. A705R	NASTY	25.3	26.3
4. A852R	QUIET	7.9	8.2
5. A965D	NASTY	36.1	35.5
6. A572C	FRIENDLY	9.6	7.7
7. A631D	FRIENDLY	11.5	27.5

Figure 2.1 A Typical Table

```
NAME OF ROW 6 = A572C <GO>
NAME OF ROW 7 = A631D <GO>
NAME OF ROW 8 = NEXT <GO>
COL 1 HEADING = PERSONALITY <GO>
COL 1 ROW 1 = FRIENDLY <GO>
COL 1 ROW 2 = QUIET <GO>
COL 1 ROW 3 = NASTY <GO>
COL 1 ROW 4 = QUIET <GO>
COL 1 ROW 5 = NASTY <GO>
COL 1 ROW 6 = FRIENDLY <GO>
COL 1 ROW 7 = FRIENDLY <GO>
COL 2 HEADING = DOSE (MG) <GO>
COL 2 ROW 1 = 8.2 <GO>
COL 2 ROW 2 = 13.2 <GO>
COL 2 ROW 3 = 25.3 <GO>
COL 2 ROW 4 = 7.9 <GO>
COL 2 ROW 5 = 36.1 <GO>
COL 2 ROW 6 = 9.6 <GO>
COL 2 ROW 7 = 11.5 <GO>
COL 3 HEADING = TAIL (CM) <GO>
COL 3 ROW 1 = 7.9 <GO>
COL 3 ROW 2 = 24.3 <GO>
COL 3 ROW 3 = 26.3 <GO>
COL 3 ROW 4 = 8.2 <GO>
COL 3 ROW 5 = 35.5 <GO>
COL 3 ROW 6 = 7.7 <GO>
COL 3 ROW 7 = 27.5 <GO>
COL 4 HEADING = EXIT <GO>
```

2.2.2 Making a Table from a File

In the PM Module, it is particularly important for the user to be able to make a table from data that is in an existing file. For this purpose, there are two types of commands available, commands for opening, closing and otherwise manipulating PM data files and commands for reading and writing these files.

The general form of the command to open a file is:

```
OPEN FILE filename
```

The general form of the command to close a file is:

```
CLOSE FILE filename
```

The general form of the command to make a table from a file is:

```
MAKE TABLE tablename FROM filename
```

The general form of the command to make a file from a table is:

```
MAKE FILE filename FROM tablename
```

Each of these commands invokes a dialogue specifying the portions of the table or file to read and write.

2.2.3 Changing Data in a Table

The DAT subsystem provides a simple mechanism for making changes to the data in a table. In order to change the dose for armadillo A631D in table ARMA (Figure 2.1), the user would use the CHANGE command as follows:

```
#CHANGE ARMA <GO>
ROW? 7 <GO>
COL? 2 <GO>
VALUE? 17.5 <GO>
ROW? EXIT <GO>
```

The resulting table would then be as shown in Figure 2.2. Notice that the value in the last row of column 2 has been changed from 11.5 to 17.5.

2.2.4 Displaying a Table

The DISPLAY command is used to display a table. The command that would display the table ARMA as shown in Figures 2.1 and 2.2 is:

```
#DISPLAY ARMA <GO>
```

2.2.5 Adding and Deleting Data in an Existing Table

The ADD COLS and ADD ROWS commands are used to add data to an existing table. Executing one of these commands re-enters the MAKE TABLE dialogue. For example, adding a column of scale shapes to the table ARMA is performed via the following dialogue:

```
#ADD COLS TO ARMA <GO>
COL 4 HEADING = SCALE
SHAPE <GO>
COL 4 ROW 1 = TRIANGLE <GO>
COL 4 ROW 2 = TRIANGLE <GO>
COL 4 ROW 3 = TRIANGLE <GO>
COL 4 ROW 4 = TRIANGLE <GO>
COL 4 ROW 5 = SQUARE <GO>
COL 4 ROW 6 = DIAMOND <GO>
COL 4 ROW 7 = DIAMOND <GO>
```

ARMA 3C X 7R ARMADILLO TAIL-GRO DATA

	1 PERSONALITY	2 DOSE (MG)	3 TAIL (CM)
1. A352C	FRIENDLY	8.2	7.9
2. A601D	QUIET	13.2	24.3
3. A705R	NASTY	25.3	26.3
4, A852R	QUIET	7.9	8.2
5. A965D	NASTY	36.1	35.5
6. A572C	FRIENDLY	9.6	7.7
7. A631D	FRIENDLY	17.5	27.5

Figure 2.2 A Typical Table Revised

```
COL 5 HEADING = EXIT <GO>
```

This new column of data could be displayed using the table portion form of the DISPLAY command:

```
#DISPLAY COL 4 OF ARMA <GO>
```

and the table portion would be displayed as shown in Figure 2.3.

Similarly, data on an additional set of 15 armadillos would be added via the ADD ROWS command. These new rows of data could be displayed via the command:

```
#DISPLAY ROWS 8 TO 22 OF ARMA <GO>
```

and the table portion would be displayed as shown in Figure 2.4.

Data may be deleted from an existing table via the DELETE command. It turns out that rows 6 and 17 of table ARMA are identical. Row 17 may be deleted using the following command:

```
#DELETE ROW 17 OF ARMA <GO>
```

2.3 Advanced Table Concepts

Some advanced table concepts are presented in this section, the most important of which is the notion of table portions. An alternate notation for tables is also presented, along with information on renaming and deleting tables.

2.3.1 Table Portions

One of the most powerful concepts that is available to the DAT subsystem user is the table portion. A table portion is a rectangular subset of the cells of a table. It may be described by specifying certain rows, certain columns, or certain cells of a table. Here are some examples of table portions used to display only part of a table of data:

```
DISPLAY ROWS 1, 3 TO 5, 8 OF DATATAB <GO>
DISPLAY COLS 5 TO 9, 1 OF DATATAB <GO>
DISPLAY ROWS 1, 2 OF COLS 6, 9 OF DATATAB <GO>
DISPLAY COL 2 OF ROWS 1 TO 3 OF DATATAB <GO>
```

The statements shown above specify a portion of the table named DATATAB by listing the rows, columns, or both which are to be included. The four ways to specify such a table portion are:

ARMA 3C X 7R ARMADILLO TAIL-GRO DATA

	4 SCALE SHAPE
1. A352C	TRIANGLE
2. A601D	TRIANGLE
3. A705R	TRIANGLE
4. A852R	TRIANGLE
5. A965D	SQUARE
6. A572C	DIAMOND
7. A631D	DIAMOND

Figure 2.3 A New Column of Data

ARMA 4C X 22R ARMADILLO TAIL-GRO DATA

	1 PERSONALITY	2 DOSE (MG)	3 TAIL (CM)	4 SCALE SHAPE
8. A536R	SICK	24.7	31.	SQUARE
9. A867R		31.2	33.8	SQUARE
10. A945C	SICK	8.9	8.9	TRIANGLE
11. A856R	QUIET	9.3	7.8	DIAMOND
12. A978D	NASTY	29.1	32.9	DIAMOND
13. A877R	DUMB	14.8	34.1	DIAMOND
14. A938C	QUIET	36.2	35.6	DIAMOND
15. A937R	FRIENDLY	40.2	36.9	SQUARE
16. A749C	NASTY	35.2	35.2	SQUARE
17. A572C	FRIENDLY	9.6	7.7	DIAMOND
18. A462C	SICK	7.8	8.	TRIANGLE
19. A465R	QUIET	9.2	8.3	DIAMOND
20. A465D	DUMB	17.9	30.9	TRIANGLE
21. A114R	FRIENDLY	46.1	38.7	SQUARE
22. A593D	NASTY	34.1	34.8	SQUARE

Figure 2.4 New Rows of Data

```
(1) ROW[S] <list> OF tablename
(2) COL[S] <list> OF tablename
(3) ROW[S] <list> OF COL[S] <list> OF tablename
(4) COL[S] <list> OF ROW[S] <list> OF tablename
```

The "<list>" construct contains numeric values or entries of the form:

```
<numeric-value> TO <numeric-value>
```

Entries in the list are separated by commas.

In addition to extracting a table portion by specifying row and column numbers, table portions may also be selected by conditional tests of values. The format for this more powerful version of a table portion is:

```
tablename WHERE <expression>
```

An example of the use of this feature is included in the following section.

2.3.2 Extracting Table Portions

The MAKE TABLE command may be used in conjunction with a table portion to extract a table portion from an existing table and make a new table from it. For example, if one wanted to examine the armadillo data in just the cases involving high doses, the following commands could be used to make a new table and display it:

```
#MAKE TABLE HIGHDOSE FROM ARMA WHERE COL 2 GT 10 <GO>
#DISPLAY HIGHDOSE <GO>
```

The resulting table HIGHDOSE is shown in Figure 2.5.

2.3.3 Setting Table Portions

The SET command can be used to change the value of a variable (see Section 2.1.2), or a single cell in a table, or all the cells in a table portion. For example the commands:

```
#SET ROW 3 COL 2 OF DATATAB TO 2.71828
#SET ROW 2 COL 3 OF DATATAB TO ROW 3 COL 2 OF DATATAB
```

would set row 3 column 2 and then row 2 column 3 of table DATATAB to 2.71828. Similarly, the commands:

```
#SET COL 5 OF DATATAB TO 1.414 <GO>
```

HIGHDOSE 4C X 14R

	1 PERSONALITY	2 DOSE (MG)	3 TAIL (CM)	4 SCALE SHAPE
1. A601D	QUIET	13.2	24.3	TRIANGLE
2. A705R	NASTY	25.3	26.3	TRIANGLE
3. A965D	NASTY	36.1	35.5	SQUARE
4. A631D	FRIENDLY	17.5	27.5	DIAMOND
5. A536R	SICK	24.7	31.	SQUARE
6. A867R		31.2	33.8	SQUARE
7. A978D	NASTY	29.1	32.9	DIAMOND
8. A877R	DUMB	14.8	34.1	DIAMOND
9. A938C	QUIET	36.2	35.6	DIAMOND
10. A937R	FRIENDLY	40.2	36.9	SQUARE
11. A749C	NASTY	35.2	35.2	SQUARE
12. A465D	DUMB	17.9	30.9	TRIANGLE
13. A114R	FRIENDLY	46.1	38.7	SQUARE
14. A593D	NASTY	34.1	34.8	SQUARE

Figure 2.5 Extracting a Table Portion

would set all the cells in column 5 of table DATATAB to 1.414.

2.3.4 Sorting a Table

The DAT subsystem has a special command for sorting a table by the data in a specified row or column. The SORT command sorts in ascending order unless the user types DESCENDING at the end of the command. For example, the following command would sort the data in table HIGHDOSE by the dose amount and display the result:

```
#SORT HIGHDOSE BY COL 2 <GO>
#DISPLAY HIGHDOSE <GO>
```

The resulting table is shown in Figure 2.6.

2.3.5 An Alternative Notation for Tables

The DAT subsystem recognizes ´TBL´[1,2] as an alternative notation for specifying the cell ROW 1 COL 2 OF TBL. Thus, one may think of a table as a two-dimensional array, and use this alternative notation to address the data items.

2.3.6 Directory of Tables

The names and sizes of all the tables that are created are held in a special table called the DIRECTORY. To get a complete list of the directory, the user can type:

```
#DIS DIRECTORY <GO>
```

A typical directory is shown in Figure 2.7.

2.3.7 Renaming and Deleting Tables

It is a good idea to keep the DIRECTORY "cleaned-up", by giving tables short but meaningful names, and by deleting unwanted tables. To rename a table, the RENAME command is used:

```
#RENAME TABLE FOO TO ´STATS´ <GO>
```

To delete a table, the DELETE command is used:

```
#DELETE TABLE <table name> <GO>
```

This will remove the table from the directory, and delete the corresponding file on the disk. Once the table is deleted, the data is lost.

HIGHDOSE   4C X 14R

	1 PERSONALITY	2 DOSE (MG)	3 TAIL (CM)	4 SCALE SHAPE
1. A601D	QUIET	13.2	24.3	TRIANGLE
2. A877R	DUMB	14.8	34.1	DIAMOND
3. A631D	FRIENDLY	17.5	27.5	DIAMOND
4. A465D	DUMB	17.9	30.9	TRIANGLE
5. A536R	SICK	24.7	31.	SQUARE
6. A705R	NASTY	25.3	26.3	TRIANGLE
7. A978D	NASTY	29.1	32.9	DIAMOND
8. A867R		31.2	33.8	SQUARE
9. A593D	NASTY	34.1	34.8	SQUARE
10. A749C	NASTY	35.2	35.2	SQUARE
11. A965D	NASTY	36.1	35.5	SQUARE
12. A938C	QUIET	36.2	35.6	DIAMOND
13. A937R	FRIENDLY	40.2	36.9	SQUARE
14. A114R	FRIENDLY	46.1	38.7	SQUARE

Figure 2.6 Sorting a Table

2C X 15R

	1 TABLE	2 SIZE
5	ARMA	2
6	AXES OF ARMAPLOT	2
7	CURVES OF ARMAPLOT	2
8	DATA OF ARMAPLOT	2
9	DATATAB	44
10	AXES OF HIPLOT	2
11	CURVES OF HIPLOT	2
12	DATA OF HIPLOT	2
13	TBL	3
14	HIGHDOSE	2
15	STATS	4

Figure 2.7 A Directory

2.4 Graphs

The Dat subsystem provides a powerful yet convenient mechanism for creating two-dimensional graphs from data which is stored in tables.

2.4.1 Example of a Graph

Figure 2.8 is an example of a simple graph. The data for this graph was extracted from the data in columns 2 and 3 of table ARMA, which is displayed in Figure 2.2.

2.4.2 Making a Graph from a Table

This graph was constructed using the MAKE GRAPH command. This command prompts the user for the title of the table, the x and y axis labels, the x and y axis forms (linear or log), the sources of the x and y values, the curve symbols, and curve labels.

The MAKE GRAPH dialogue that led to the graph in Figure 2.8 is given below:

```
#MAKE GRAPH ARMAPLOT <GO>

TITLE OF GRAPH? EFFECT OF TAIL-GRO ON TAIL LENGTH <GO>
X AXIS LABEL? DOSE(MG) <GO>
X AXIS LINEAR(LIN) OR LOG? LIN <GO>
Y AXIS LABEL? TAIL
LENGTH <GO>
Y AXIS LINEAR(LIN) OR LOG? LIN <GO>
X VALUES FROM TABLE(T) OR ENTER(E)? T <GO>
ENTER TABLE PORTION: COL 2 OF ARMA <GO>
ENTER SOURCE OF CURVE 1: TABLE(T) OR FUNCTION(F)? T <GO>
ENTER TABLE PORTION: COL 3 OF ARMA <GO>
CURVE SYMBOL? SQUARE <GO>
CURVE LABEL? ARMADILLO DATA OF 1/6/79 <GO>
CONNECTED -- YES(Y) OR NO(N)? N <GO>
EXIT(E), NEW X-VALUES(X) OR Y-VALUES(Y) FOR NEXT CURVE: E <GO>
```

2.4.3 Displaying a Graph

The DISPLAY command is used to display a graph. The command that would display the graph ARMAPLOT as shown in Figure 2.8 is:

```
#DISPLAY ARMAPLOT <GO>
```

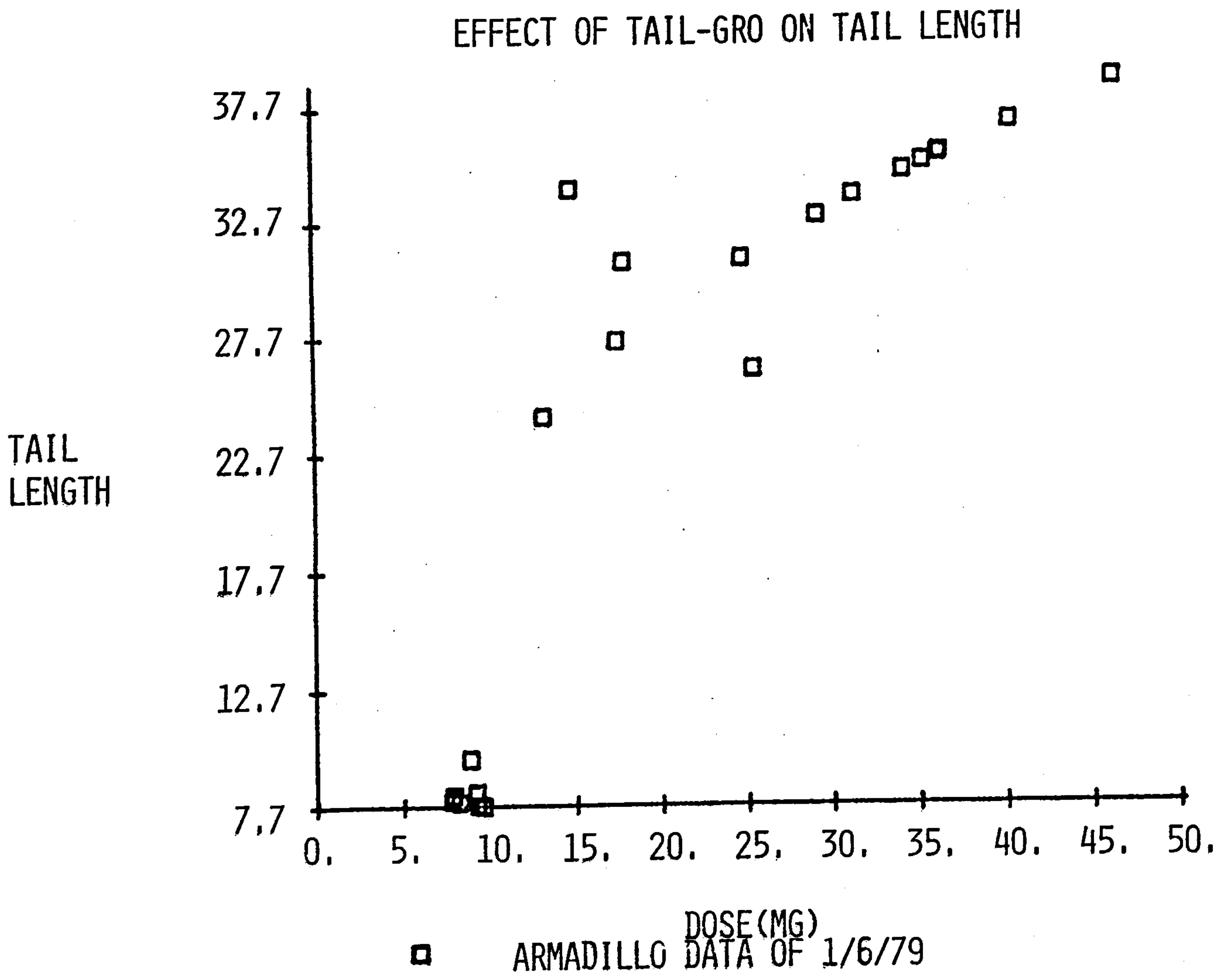


Figure 2.8 A Simple Graph

2.4.4 Editing a Graph

The DAT subsystem does a reasonable job in selecting the proper defaults for upper and lower limits on axes etc. However, there are a number of parameters which may be changed through the graph editing commands. There are two sets of parameters which can be changed by the editing commands: axis information and curve information. There are five variables which may be set for each axis. The table below shows the name of the variables and their assigned values:

VARIABLE	**VALUE**	**USE**
LABEL	<text>	name of axis
LINLOG	LIN or LOG	linear or log axis
LOW	<numeric value>	lowest value shown
HIGH	<numeric value>	highest value shown
UNITS	<numeric value>	units per tick

These variables may be TYPEd or SET like any other DAT subsystem variable.

There are five variables which pertain to the curves:

VARIABLE	**VALUE**	**USE**
LABEL	<text>	in legend
SYMBOL	<letter or plotting symbol>	plotting symbol
CONNECTED	YES or NO	connected by lines?
LINE	<text> (see below)	type of connecting line
FUNCTION	<text>	

The LINE specification can be one of the following:

SINGLE
DOUBLE
DOT
DASH
DOT DASH
DOT DOT DASH

FUNCTION accepts an expression in X from which it generates a set of Y values, e.g.

```
´X**2 + SIN(X/2)´
```

The curve variables may be accessed using the following syntax:

```
SET <variable> OF CURVE <number> OF <graph name> TO <value>
```

When a graph is first constructed, these variables are all given values.

In the following dialogue, the user adjusts the LOW and HIGH values of ARMAPLOT and then DISPLAYs the revised graph:

```
#SET LOW OF Y AXIS OF ARMAPLOT TO 0 <GO>
#SET HIGH OF Y AXIS OF ARMAPLOT TO 40 <GO>
#DIS ARMAPLOT <GO>
```

The revised graph is shown in Figure 2.9.

2.4.5 Adding and Deleting Curves

Once a graph has been constructed, curves may be added or deleted. Deleting curves from a graph is similar to deleting rows or columns from a table. The user simply types:

```
#DELETE CURVE[S] <list> OF <graph name> <GO>
```

The list or curves is a list of curve numbers of the curves to be deleted. The curves are numbered in the order they are listed in the legend below the graph, starting with 1.

Adding curves to a graph is similar to making the graph in the first place. The user simply types:

```
#ADD CURVE[S] TO <graph name> <GO>
```

This will re-enter the make graph dialogue at the point where the X specifications are determined for a curve.

Any curve additions or deletions have a permanent effect. The original graph will have been altered. Subsequent displays of the graph will show the new form.

2.4.6 Fitting Polynomials to Graphs

The DAT subsystem allows one to fit a polynomial to a curve of data points on a graph. The result is the addition of a function curve to the graph. To fit a straight line to a curve, the FIT LINE command may be used; to fit a polynomial to a curve, the FIT POLYNOMIAL command is used. Figure 2.10 is a graph called GROPLOT containing a curve of data points to which a straight line has been fit via the following command:

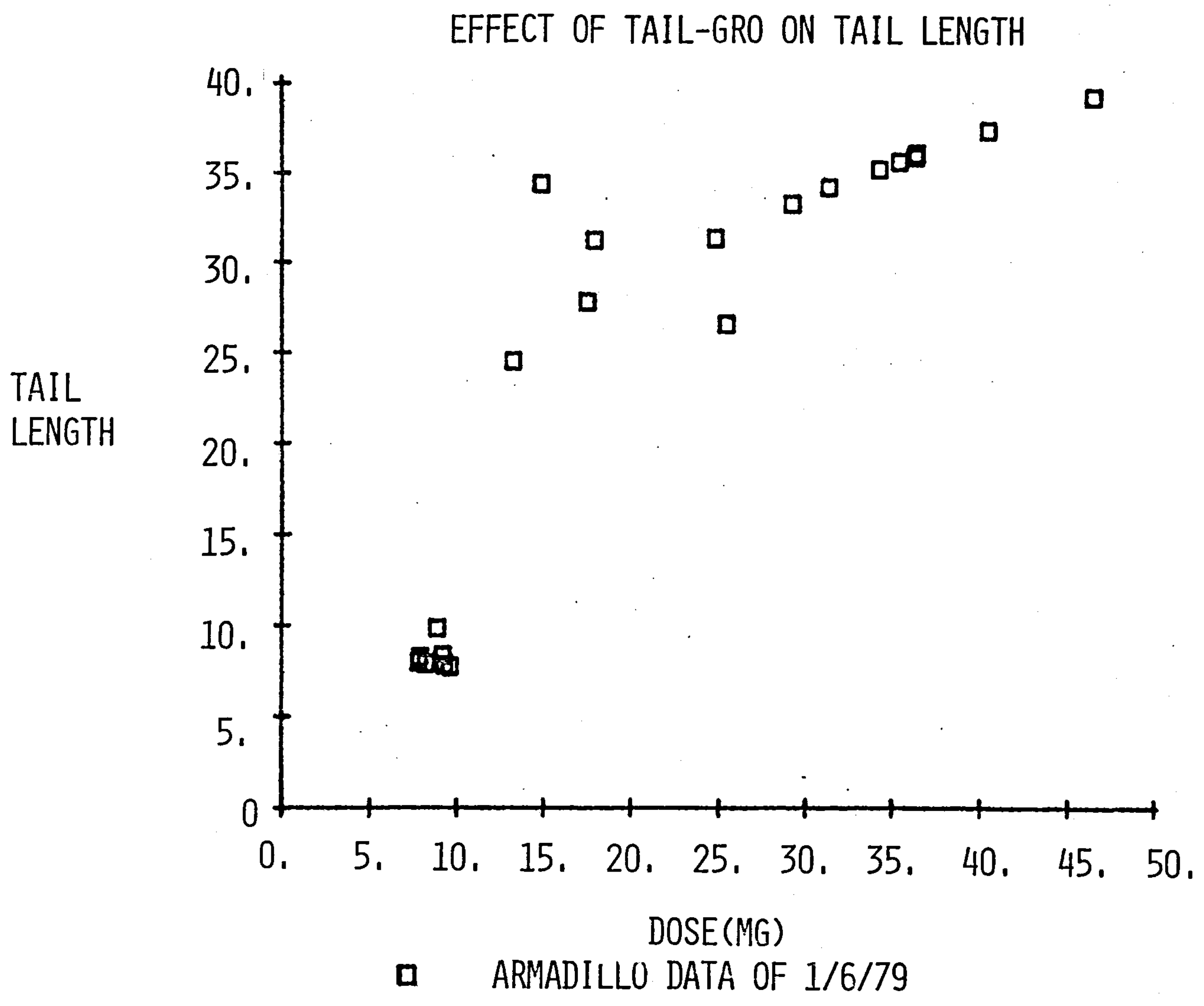


Figure 2.9 A Simple Graph Revised

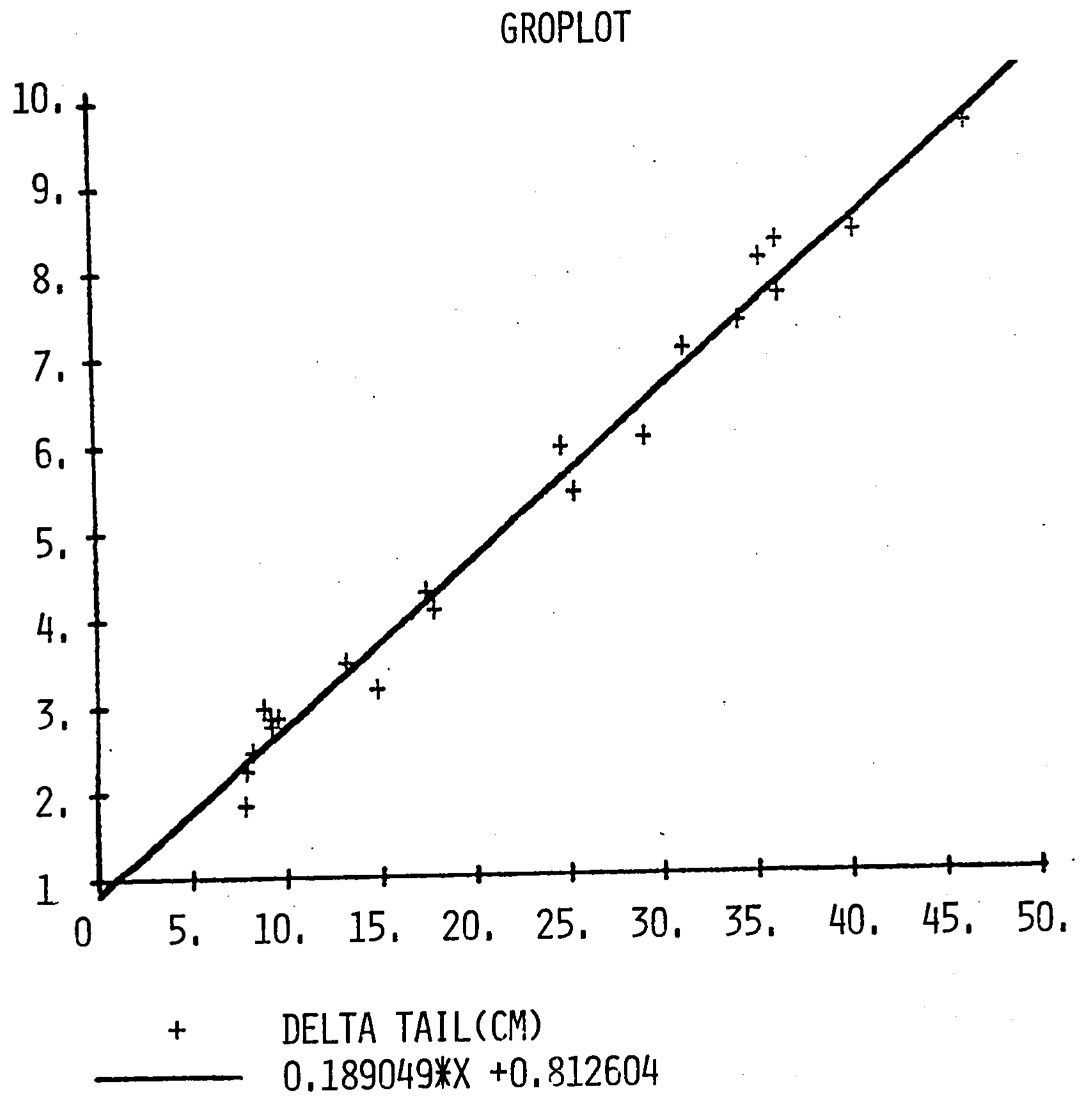


Figure 2.10 A Polynomial Fit to a Graph

```
    #FIT LINE TO CURVE 1 OF GROPLOT
```

2.5 Using DAT Subsystem Procedures

DAT subsystem procedures are written in the DAT programming language which is described in the next section. These procedures may be invoked either directly at the DAT command level or within another procedure.

There are two methods of invoking a procedure: (1) the CALL statement, e.g.

```
    #CALL GETLINE(STRING,CHAN) <GO>
    #CALL ERASE <GO>
```

and (2) as a function call, e.g.

```
    #SET X TO SIN(Y) <GO>
    #X = SIN(Y) <GO>
    #TYPE COS(Y) <GO>
```

The call statement is used if the procedure does not return a result. Such a procedure is run for its side-effect, such as changing the value of its arguments or performing input or output.

2.5.1 Example of a Procedure

SPIRO is a procedure that draws fancy spirals, such as shown in Figure 2.11. This figure was produced by calling SPIRO as follows:

```
    #CALL SPIRO(123,123,1.02,1000) <GO>
```

The procedure SPIRO is written in the DAT programming language; its code is shown in Figure 2.12. Detailed information on the DAT programming language is presented in the next section.

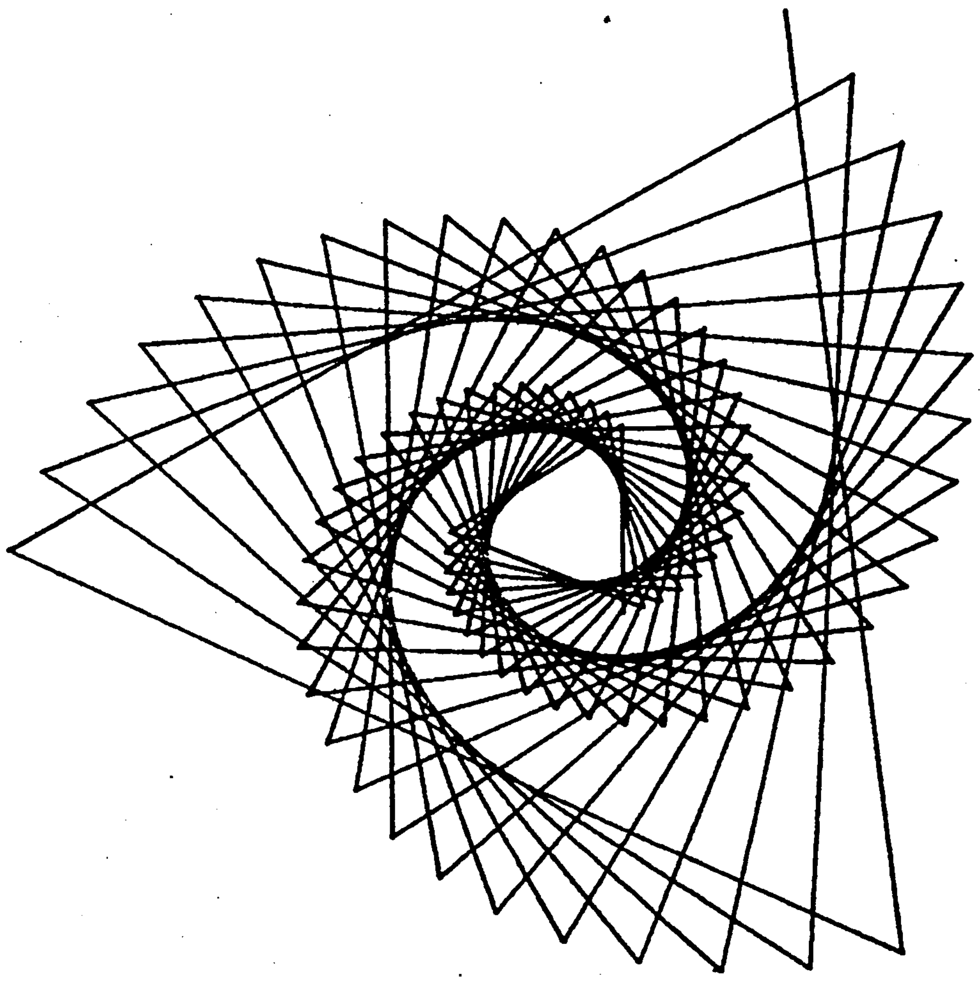


Figure 2.11 A Spiral Drawn by Procedure SPIRO

```
<TOP OF TEXT>
/* SPIRO */
PROCEDURE(RAD,ANGL,ATT,NUM,X0,Y0);
/* THIS PROCEDURE DRAWS FANCY SPIRALS */

CALL ERASE;                       /* CLEAR THE SCREEN */
ANG=0;

IF NARGS=4                        /* OPTIONAL ARGUMENTS */
THEN DO; X0=512; Y0=390; END;

DO I = 1 TO NUM;                  /* LOOP NUM TIMES */
   X1=RAD*COSD(ANG)+X0;
   Y1=RAD*SIND(ANG)+Y0;
   IF X1>1023 OR Y1>780 OR X1<0 OR Y1<0
   THEN DOEXIT;                   /* EXIT IF OFF SCREEN */

   CALL DLINE(X0,Y0,X1,Y1);       /* DRAW THE LINE */

   X0=X1;                         /* UPDATE VARIABLES */
   Y0=Y1;
   RAD=RAD*ATT;
   ANG=MOD(ANGL+ANG,360);
END;

END;
<BOTTOM OF TEXT>
```

Figure 2.12 Procedure SPIRO

3. The DAT Programming Language

3.1 Introduction to the DAT Programming Language

The DAT programming language permits the user to create new procedures. There are two major purposes for these user-defined procedures. First, they enable the user to automate a sequence of analysis steps, such as reading a file, computing some measures, and plotting the results. Second, using the DAT programming language, the user can conveniently create and modify new performance measures and try them out on actual data.

The DAT programming language is an integrated part of the DAT subsystem. Thus, DAT commands and DAT programming statements are interchangeable; programming statements may be used at the DAT command level, and DAT commands may be employed within procedures. In addition, the data (tables and graphs) are accessible from either the DAT command level or from within procedures. Whatever manipulation can be done at the command level, can also be done automatically, within a procedure.

3.2 Data Structures

All of the DAT subsystem data types and data structures are accessible from the DAT programming language.

3.2.1 Constants, Variables and Expressions

There are four types of data values available in the DAT programming language: FIXED numbers, FLOAT numbers, TEXT values, and an EMPTY value. These have the same meanings and uses as in the DAT subsystem, and are described in Section 2.1.3.

Variables are data items that can change in value, unlike constants, whose values are explicit and unchangeable. Any variable may be assigned any of the four types of data values. Variables used in a procedure should be assigned a value before they are referenced in a statement or expression. This process is known as initializing a variable. An uninitialized variable is considered to be EMPTY. Variables do not have to be "declared" before they are used, however; space is allocated to a variable when it is used for the first time.

A variable is referred to by a variable name. This name may consist of an arbitrarily long string of characters, the first of which must be alphabetic.

Values may be computed by means of expressions. An expression is formed by using operators and functions to combine constants and variables. The expressions, operators and functions available are the same as in the DAT subsystem, and are described in Section 2.1.4

3.2.2 DAT Tables

The data tables available in the DAT subsystem are also available in the DAT programming language. In the DAT programming language, however, tables are made either from a file, or from an existing table portion. The MAKE TABLE dialogue is not available. One could create a table by MAKEing a table, and assigning the row and column names, and data values via the SET statement. Similarly, the CHANGE TABLE command is not available; one could merely SET the data values to new values. In general, the DAT subsystem commands which produce a dialogue are not available; the rest are available.

3.2.3 Graphs

The graphs available in the DAT subsystem are also available in the DAT programming language. As with tables, however, the MAKE GRAPH dialogue is not available. One would create a graph by MAKEing a graph from a table, and assigning the title, axis labels, etc. via the SET command. Similarly, the dialogue following the DELETE CURVE and ADD CURVE commands are not available; one would merely SET the parameter values. In general, as with tables, the DAT subsystem commands which produce a dialogue are not available; the rest are available.

3.3 Programming Statements

The set of DAT programming statements is small and powerful. There are only six types of statements:

assignment statements,
input/output statements,
conditional statements (IF-THEN-ELSE),
block and loop constructs (DO-END block, DO loop, DO-WHILE),
unconditional jumps (GO TO).
and procedure calls and returns,

All DAT procedures (programs) are constructed from combinations of these six elements, plus any of the command-level The first five types of statements are covered in this section, and the last, procedure calls and returns are covered in the next section 3.4. commands.

In general, DAT programming statements can be written in any format. Statements can be broken by carriage returns or by multiple spaces between words. At least one space is required between separate words, and each statement must be terminated with a special character

- a semicolon (";"). The semicolon is analogous to the <GO> character at the command level.

3.3.1 Assignment Statements

Assignment statements may take one of two forms. Either the SET statement is used, or the "=" operator is used. The general form of the SET statement is:

```
SET variable TO expression;
```

An assignment statement using the "=" operator takes the form:

```
variable = expression;
```

3.3.2 Input/Output Statements

Input and output statements constitute the means of getting data in or out of DAT procedures. The basic I/O elements are extremely simple. The INPUT statement is used for input, and the TYPE statement for output.

The basic input statements in the DAT programming language are:

```
INPUT numeric-variable;

INPUT TEXT text-variable;
```

An input statement does two things: (1) It causes the program to pause and await the entry of a value to be entered by the user via the keyboard; and (2) it sets the variable named in the INPUT statement to the value entered.

Numeric values are entered with the INPUT statement. The data supplied to an INPUT statement must be a number, terminated with the <GO> key. The INPUT statement checks that the value is either a FLOAT or a FIXED, and returns an error message if it is not.

TEXT values are input with the INPUT TEXT statement. Any text entered will be taken as the value of the text variable. The text must be terminated with the <GO> key. If no text is typed (i.e., just <GO>) the text-variable will have the value of the null text (''), not EMPTY. Upon responding to an INPUT TEXT prompt, the text string need not be quoted; the DAT programming language does this implicitly.

All of the normal DAT subsystem display commands are valid within the DAT programming language, and they comprise the best way to output such variables as tables and graphs. For numbers and text, there are two simple statements:

```
TYPE variable;

TYPE NOCR variable;
```

The first statement types out the value of any variable followed by a carriage return. The TYPE NOCR outputs the variable without the carriage return.

The TYPE statement can output an arbitrarily long list of variables, if the variable names are separated by commas. A carriage return will be typed after each value.

3.3.3 Conditional Statements

A conditional statement is one that tests a condition and executes a command based on the results of the test. The basic conditional construct is the IF...THEN statement, and its adjunct, the ELSE statement.

The IF...THEN statement can be formally expressed as:

```
IF boolean-expression THEN statement;
```

where "boolean-expression" is any expression (or single variable) which evaluates to TRUE or FALSE. Any legitimate expression containing relational or boolean operators meets this criterion, as does any function that yields a truth-value.

The "statement" portion of the IF...THEN construction may be any legal DAT programming language statement that can be executed, including another IF...THEN statement, or a DO...END block (see 3.3.4).

When the boolean-expression in an IF...ELSE statement is evaluated as FALSE, the THEN statement is not executed. When the boolean-expression is TRUE, the THEN statement is executed.

The ELSE statement is an extension of the IF...THEN statement that increases its power. It must immediately follow an IF...THEN statement. The general form of the IF...THEN...ELSE construction is:

```
IF boolean-expression THEN statement;
ELSE [statement];
```

where the statement following the ELSE is optional. The effect of the ELSE statement is to specify an alternative action to be performed when the IF condition is FALSE. The statement following the ELSE is optional so that nested IFs and ELSEs may be paired even when some ELSE alternatives do not exist.

3.3.4 Block and Loop Constructs

The DO...END block combines a group of statements into constructs that are treated logically as a single statement. Here is the general form of the DO...END block:

```
     DO;
             statement;
             statement;
                 .
                 .
                 .
             statement;
     END;
```

where "statement" is any executable statement, including both simple and nested conditional statements.

One of the most important uses of the DO...END block is to program a series of operations, based on the outcome of a conditional statement. For example:

```
     IF NOTEMPTY(X)
        THEN DO;
                 TYPE X;
                 SUMM=SUMM+X;
             END;
        ELSE TYPE 'EMPTY VALUE';
```

A DO...END block can also be used following an ELSE statement.

The DO loop is a special form of the DO...END block which allows a group of statements to be executed in an iterative way. DO loops may be further classified into two distinct types: the basic DO loop and the DO WHILE loop. The basic DO loop repeats a set of operations a number of times, with the number of iterations specified in the DO statement. The DO WHILE loop is more open-ended; it continues its iterations an unspecified number of times, repeating the set of operations as long as the specified condition holds.

The general form of the DO loop is as follows:

```
     DO counter = start-value TO quit-value [BY increment];
        statement;
        statement;
            .
            .
            .
        statement;
     END;
```

"Counter" is a numeric variable that keeps count of the number of times the loop is repeated. "Start-value" and "quit-value" specify the number of iterations. The counter proceeds from the start-value to the quit-value by an increment of 1 each time the loop is executed, unless a different increment is specified in the "BY increment" option.

The general form of the DO...WHILE loop is as follows:

```
DO WHILE (boolean-expression);
     statement;
     statement;
          .
          .
          .
     statement;
END;
```

where boolean expression is any expression that evaluates to TRUE or FALSE. The parentheses around the boolean-expression are required. The loop will continue to iterate until control returns to the first line and the boolean-expression is evaluated as FALSE.

For greater flexibility, DO loops and DO...WHILE loops can be terminated before the normal quitting condition is reached by means of the DOEXIT and DONEXT statements. When a DOEXIT statement is encountered in a loop, control immediately passes out of the loop to the statement following the END. When a DONEXT statement is encountered in a loop, control jumps back to the first line of the loop and the next iteration is performed.

3.3.5 Unconditional Jumps

A GO TO statement causes control to jump to a statement with a specified label. The general form of the GO TO construct is:

```
GO TO label;
     statement;
     statement;
          .
          .
          .
     statement;
label: statement;
```

where "label" is a legal name, meaning that it must be a legal variable name. Any statement, except and END statement can have a label. Labels must be unique within a single procedure, although more than one GO TO statement can refer to a single label.

When a GO TO statement is executed, control is transferred to the statement marked by the label specified in the GO TO.

3.4 Procedures

Procedures in the DAT programming language are analogous to subroutines in FORTRAN. A procedure has a name by which it can be referenced, and allows for arguments to be passed to and from it.

The three statements which make procedures usable are PROCEDURE, CALL and RETURN.

3.4.1 The PROCEDURE Statement and Arguments

The PROCEDURE statement delimits and identifies a block of code as a functional unit. The end of the procedure is delimited with a corresponding END statement. The general form of a procedure is:

```
PROCEDURE procedure-name [(arguments)];
     statement;
     statement;
          .
          .
          .
     statement;
END;
```

The "procedure-name" is the name given when CALLING a procedure. The "arguments" are data values that are supplied to the procedure, upon which the procedure operates. Within a procedure definition, arguments are "dummy" variables having no value until the procedure is called. When the procedure is called, the constants, variables or expressions given in the CALL statement are substituted for the dummy statements in the PROCEDURE statement. The procedure then operates on the real values. The CALL statement is discussed more fully in section 3.4.3. The arguments in a PROCEDURE statement must be enclosed in parentheses, and if there are more than one argument, they must be separated by commas.

3.4.2 The RETURN Statement

The RETURN statement is used to exit from a procedure. There may be more than one RETURN statement in a procedure. The general form of the RETURN statement is:

```
RETURN [value];
```

where the optional "value" may be a constant, variable, or expression. If the value is absent, the RETURN statement causes an immediate exit from the procedure in which it occurs, and a return to the place from which the procedure was called. The form of the RETURN statement with the value is used to transfer data from a procedure, as well as to exit a procedure; the "RETURN value" statement is used to make a procedure behave like a function. For example if the procedure FOO takes two arguments and returns a value, it could be invoked by a statement such as:

```
NEWVAL = FOO(A,B);
```

where NEWVAL is a variable to which the value returned is assigned, and A and B are the arguments supplied to FOO.

3.4.3 The CALL Statement

As noted in the previous section, a procedure that returns a value can be invoked as a function call, by setting a variable to the value of a procedure. This form of invocation cannot be used be used for procedures which do not return a value. To invoke such a procedure the CALL statement should be used. The form of the CALL statement is:

```
CALL procedure-name [(arguments)];
```

where "procedure-name" is the name given in the PROCEDURE statement defining the procedure, and the optional arguments are one or more constants, variables, or expressions, separated by commas.

3.4.4 Commenting Procedures

A procedure definition can contain statements that are not executed. These statements are called comments, and are usually used to describe the purpose and effects of the procedural statements. A comment is set off by the characters "/*" (slash, asterisk) at the beginning and "*/" at the end. These symbols and any text that occurs between them are ignored and never executed. Any characters may be included within the comments delimiters but the delimiters themselves. Comments may continue through any number of lines, and may be placed anywhere in the definition, except in the middle of a single word.

3.5 Building a Procedure

In the previous section, the definitive features of procedures, their uses, and the means by which they are invoked were discussed. In this section, the creation of procedures is discussed: how to enter them into the DAT subsystem.

3.5.1 Editing Basics

The DAT subsystem includes a simple editor which is used to edit procedure definitions. The editor is a "line-oriented" editor, similar to the EDIT program in use at LRC.

To initiate editing of a procedure, the user types at command level:

```
#EDIT PROCEDURE procedure-name <GO>
```

where "procedure-name" can be the name of an existing procedure, which is to be modified; or it can be the name of a new procedure, as yet undefined.

If it is ordinary text that is being created or edited and not the definition of a procedure, the command would be:

```
#EDIT TEXT text-name <GO>
```

where "text-name" is the variable name to use for the TEXT constant being edited. To have a PROCEDURE definition typed out at the terminal, the command to use is:

```
#TYPE DEFINITION OF procedure-name <GO>
```

or

```
#TYPE DEF OF procedure-name <GO>
```

On the other hand, to have a TEXT constant typed out at the terminal, the command to use is merely:

```
#TYPE text-name <GO>
```

Once the EDIT command has been given to enter a new text, the EDITOR responds by displaying the following:

```
0 <TOP OF TEXT>
1
2 <BOTTOM OF TEXT>
```

If the text of a short procedure were to be input, using the editing commands listed below, the EDITOR might display:

```
0 <TOP OF TEXT>
1 PROCEDURE (A);
2 TYPE A;
3 END;
4
5 <BOTTOM OF TEXT>
```

Only the three statements on lines 1-3 are in the actual definition. The new line numbers are added automatically. The EDITOR updates and redisplays the text (renumbering the lines) after each editing command. For example, if the statement "A = A+1; <carriage-return>" were to be added after the PROCEDURE statement, the screen would display:

```
0 <TOP OF TEXT>
1 PROCEDURE (A);
2 A = A+1;
3 TYPE A;
4 END;
5
6 <BOTTOM OF TEXT>
```

The EDITOR displays at most 25 lines of text on the screen; which lines are on the screen are under the user's control via the screen commands discussed below. When lines are inserted into a full screen, the lower lines move off the bottom of the screen to accommodate the new lines above. The EDITOR does not permit editing ay off-screen lines. To edit a line, it must first be displayed on the screen.

3.5.2 EDITOR Commands

The EDITOR permits four kinds of actions:

(1) **FORWARD** and **BACK** permit the user to control which portion of a long text appears on the screen;

(2) **INSERT**, **KILL**, and **CHANGE** permit line-at-a-time changes to the text;

(3) **REPLACE** permits replacements within a line;

(4) **EXIT** and **ABORT** take the user from the EDITOR back to command-level.

Any action which results in a change to the text causes the screen to be redisplayed. However, unless one of the screen commands has been given, redisplay begins with the same line at the top.

Here are the detailed specifications of the commands (portions in brackets are optional):

3.5.2.1 The FORWARD Command

```
F[ORWARD] [line-count] <GO>
```

FORWARD moves text upward into the screen area. If the optional line-count is included, FORWARD will move that number of lines forward into the text. If the line-count is not included, FORWARD moves the text so that the line following the last line currently displayed will be at the top of the new screen.

3.5.2.2 The BACK Command

```
B[ACK] [line-count] <GO>
```

BACK moves the text downward into the screen area. If the line-count is not included, BACK moves the text so that the line preceding the first line currently displayed will be at the bottom of the screen.

3.5.2.3 The INSERT Command

```
I[NSERT AFTER] line-number <GO> INSERT:
     one or more lines of text <GO>
```

INSERT accepts lines of text (separated by carriage returns) and places them after line-number. The line-number argument is not optional. Note that two "<GO>"s are required -- one after the command specification (following which the EDITOR types the word "INSERT:"), and one after the last line of input text.

3.5.2.4 The KILL Command

```
K[ILL] first-line-number [[TO] last-line-number] <GO>
```

KILL deletes line first-line-number, or the lines first-line-number through last-line-number inclusive, depending upon the option specified.

3.5.2.5 The CHANGE Command

```
C[HANGE] first-line-number [[TO] last-line-number]  <GO>  INSERT:
     one or more lines of text <GO>
```

CHANGE is exactly the equivalent to a KILL followed by an INSERT. It avoids an extra erase and redisplay of the screen, and implicitly inserts the specified text beginning at the same position as the text deleted.

3.5.2.6 The REPLACE Command

```
R[EPLACE] line-number <GO> REPLACE: text-to-match <GO>
     WITH: text-to-substitute <GO>
```

Replace permits a single portion of a line to be replaced with another group of characters, without retyping the entire line. It finds the first occurrence in line line-number of the text-to-match, and replaces it with the specified text-to-substitute, then redisplays the screen.

Note that three "<GO>"s are required: one after the command specification, following which the EDITOR types "REPLACE:"; one after the text-to-match, following which the EDITOR types "WITH:"; and one after the text-to-substitute. If the text-to-substitute is omitted, the text-to-match will merely be deleted. Carriage returns may be included in either text argument, and thus reduce or increase the number of lines.

3.5.2.7 The EXIT Command

```
EX[IT] <GO>
```

EXIT transfers control back to command-level, saving the newly edited version and replacing the old version with the new.

3.5.2.8 The ABORT Command

```
ABORT <GO>
```

ABORT transfers control back to command-level, leaving the original text unchanged. No changes that have been made in the EDITOR will be preserved. ABORT cannot be abbreviated in order to prevent its accidental execution.

4. The STAT Subsystem

The purpose of the STAT subsystem is to provide a tool for the scientific investigator to perform statistical tests and make statistical inferences from his data.

As part of the design of the PM Module, a survey was made of existing, commercially available statistical analysis packages. It was hoped that such a package could be incorporated into the PM Module, thereby avoiding the needless time and expense of extensive software development. Most of the statistical packages under consideration have taken more than about five man-years to develop.

4.1 Requirements of the STAT Subsystem

In order to be considered for use as the STAT subsystem, a statistical package would have to meet the following four basic requirements:

(1) Includes a large range of **standard statistical analyses**, including analysis of variance.

(2) Runs **interactively**.

(3) Runs on a **CDC Cyber-175** computer.

(4) Provides for **data transformation**.

4.1.1 Standard Statistical Analyses

The STAT subsystem should be capable of performing many different types of statistical analysis. Some examples of classes of standard statistical analyses are the following:

(1) **Descriptive statistics.** This includes measures of central value (mean, median, mode), measures of variability (variance, standard deviation, differences between percentiles, etc.), measures of skewness, measures of the agreement with a Gaussian distribution, along with tables and plots of frequency distributions, histograms, etc.

(2) **Analysis of Variance.** This analysis is used to determine the relationship and measure the contributions of a number of independent variables with a dependent variable, where the independent variables are **categorical**. This analysis is most relevant for the PM Module. One of the most frequently used experimental paradigms in research on performance measures is to make sets of runs with one (univariate) or more (multivariate) conditions at several discrete levels.

The outcome of these runs is a set of performance measures, some of which will have a relationship to the varied conditions. Analysis of variance is the technique used to measure these relationships.

(3) **Correlation and Regression Analysis.** This analysis is also used to determine the relationship and measure the contributions of a number of independent variables with a dependent variable, but here all the variables are **continuous**. Scatter diagrams are also useful in examining these relationships.

(4) **Tests of a Model.** A variety of statistical tests may be used to determine whether a set of experimental results differs from the predictions of a model. These include t-tests, chi-squared tests, etc.

(5) **Discriminant Analysis.** This analysis is used to decide whether groups are different based on the information available in some number of analysis variables.

(6) **Factor Analysis.** This refers to a number of techniques for analyzing correlation coefficients. The goal is to discover a few basic patterns or components in the relationships found in the correlation matrix.

(7) **Other Analysis Techniques.** These techniques include non-parametric tests, et al.

4.1.2 Interactive Operation

In order to allow for the efficient interchange of information between the analyst and the computer, the STAT subsystem must be interactive. The system must allow the analyst to interrogate the data base repeatedly and rapidly examine different statistical measures of different performance measures. It is also very desireable to be able to switch to a batch mode of operation once an analysis procedure is selected for use with a large data base.

4.1.3 CDC Cyber-175

The entire PM Module must run on The CDC Cyber-175 computer at LRC. This machine runs the NOS operating system. Any statistical packages which do not run on this system at present would have to be converted. In general for a package of this size, the conversion cost would be prohibitive, if it were even possible.

4.1.4 Data Transformation

This requirement is the loosest. With a powerful data transformation capability, a statistical analysis package could fulfill virtually all the requirements of the entire PM Module. A more modest goal is to meet the requirements of the Baseline PM Module.

The basic requirements of data transformation are:

(1) **Edit data.** Experimental data often contains values which are anomalous or erroneous, and which must be edited.

(2) **Select data.** It is important to be able to specify and select a subset of the data for analysis.

(3) **Sort and Merge data.** Data from various sources often has to be combined for joint analysis. For example, several runs from one subject might be combined for a subject summary, or several runs from several subjects under one condition might be combined.

(4) **Create new variables from old.** This transformation capability would allow performance measures to be created from raw experimental results such as time series. There is virtually no limit to the degree of flexibility that would be useful here.

4.2 Candidate Statistical Analysis Packages

In the course of conducting the survey of existing, commercially available statistical analysis packages, on the order of sixty different packages received at least cursory consideration. They ranged from small, limited purpose packages to large general purpose packages; from free, unsupported software to multi-thousand dollar systems with annual upgrades; from card oriented batch systems to highly conversational systems with on-line instruction; and from non-integrated subroutine libraries to fully integrated operating systems. The primary sources of information regarding these packages were two papers: Shucany, Minton and Shannon (1972), and Anderson and Sims (1977), as well as numerous conversations with many users of statistical packages.

After a preliminary investigation of the packages, the number under consideration was reduced to the following three:

(1) **P-STAT 78**, produced by P-STAT, Inc. of Princeton, New Jersey.

(2) **SIPS**, produced by the Department of Statistics of Oregon State University, Corvallis, Oregon.

(3) **SCSS**, produced by SPSS Inc. of Chicago, Illinois.

The first of these three candidate packages to be examined in detail was SCSS. It was found to have many strong features including a highly conversational front end, reasonable data transformation capabilities, and a moderate range of available statistical analyses. It had two major shortcomings, however: (1) It would not be available for operation on the CDC Cyber computer for at least six months or perhaps a year, and (2) It did not include analysis of variance. On the basis of these two shortcomings, SCSS was not given further consideration.

The two remaining candidate packages, P-STAT 78 and SIPS were examined next. Their strength and weaknesses are discussed in the next section.

4.3 Two Prime Candidates: P-STAT 78 and SIPS

These two statistical analysis systems are compared by first considering how well they meet the four criterion listed above. Other factors, such as documentation, vendor support, and cost are then considered.

4.3.1 Standard Statistical Analyses

Both SIPS and P-STAT 78 include a wide range of standard statistical analyses, although SIPS has a significantly broader range than P-STAT 78.

(1) **Descriptive Statistics.** Both packages compute a wide variety of descriptive statistics, along with tables and plots. P-STAT 78 provides a more complete and flexible facility for formatting the output for reports.

(2) **Analysis of Variance.** P-STAT 78 contains a multi-variate analysis of variance subsystem. However, it is rather inconvenient to use; the software is not nearly up to the standards of the rest of the P-STAT package. SIPS contains adequate univariate and multivariate analysis of variance subsystems.

(3) **Correlation and Regression Analysis.** Both P-STAT 78 and SIPS include fairly comprehensive correlation and regression analyses.

(4) **Tests of a Model.** Both P-STAT 78 and SIPS include a variety of statistical tests of this type. SIPS offers a significantly wider range of tests, including several parametric and non-parametric tests.

(5) **Discriminant Analysis.** P-STAT 78 includes discriminant analysis. SIPS includes discriminant analysis a part of the MANOVA subsystem.

(6) **Factor Analysis.** Both P-STAT 78 and SIPS include commands for performing factor (principal components) analysis of a covariance matrix, and for performing rotations based on this analysis.

(7) **Other Analysis Techniques.** SIPS includes a variety of non-parametric tests such as Wilcoxon, Mann-Whitney, Kruskal-Wallis, sign test, etc.

4.3.2 Interactive Operation

Both the P-STAT 78 and SIPS systems operate interactively; both allow the user to rapidly explore a set of statistical analyses, to drop one and pursue another as the results are obtained. P-STAT 78 provides a greater level of diagnostic error messages and a more elegant scheme of error recovery. The glaring exception to this feature occurs in the P-STAT 78 Analysis of Variance routines where a simple user error can crash the entire P-STAT system.

Both the P-STAT 78 and SIPS systems provide convenient mechanisms for operating in a batch mode, as might be desired when analyzing a large data base. P-STAT provides a more convenient method for setting up the batch run; the user has the ability to edit the typescript generated in a smaller interactive run.

4.3.3 CDC Cyber-175

Both the P-STAT 78 and SIPS systems are written primarily in FORTRAN, and will operate on the CDC Cyber-175 computer at LRC. P-STAT is written in machine independent form, and has been made to run on a CDC Cyber system by means of a preprocessor conversion program. SIPS has been developed exclusively on CDC computers and has not yet been converted for use on other machines. It is highly optimized for use on the CDC Cyber computers under the NOS operating system, and should run significantly faster than P-STAT on the Cyber-175 at LRC.

4.3.4 Data Transformation

Both P-STAT 78 and SIPS provide mechanisms for fairly extensive data transformation. Although neither system would fulfill all the requirements for data transformation for the entire PM Module, either one would meet the requirements of the baseline PM Module.

Both systems allow for editing data, selecting data, and sorting and merging data. In addition, both systems allow new variable to be created from old ones. The ability to create new variables from arbitrary sets of old variables is limited.

4.3.5 Other Considerations

The P-STAT and SIPS systems have similar origins; both began as statistical analysis systems serving a university computing center. P-STAT was originally developed at Princeton in 1962, and SIPS at Oregon State University in the early 1970's. The aims and directions taken by the two systems following their initial development, however, have been somewhat different.

The emphasis in **P-STAT** has been on making the system appealing to as broad a community as possible. The following attributes of the P-STAT system result, at least in part, from this emphasis:

(1) The documentation for P-STAT is very complete and up-to-date. Each feature of the system is explained in some detail.

(2) The system works on a wide variety of computer systems (including IBM, UNIVAC, DEC, CDC, Honeywell, Burroughs, PRIME, INTERDATA, Hewlett-Packard, and Harris), although it is probably not highly optimized on many of them.

(3) A private corporation, P-STAT, Inc., has been established to sell and support the P-STAT system. Consequently, the problems encountered by users are handled effectively.

(4) The interactive front-end (the user interface) of P-STAT is quite sophisticated. When P-STAT is used interactively, an editor file is kept which contains both the commands and data records which have been typed in. This file can be accessed at any time to fix errors, or to add, delete, or replace commands or data. The corrected commands can then be re-executed. The editor file can be saved for use in another interactive session of for submission as a batch job.

(5) There is great flexibility in providing well formatted tables and graphs, which are suitable for inclusion "as-is" in reports.

(6) Although the range of available statistical analyses is large, there has not been an effort to maintain an exhaustive set of analyses.

(7) P-STAT contains commands which read and write BMDP and SPSS system files. Thus, the P-STAT user has access to two of

the most powerful and widely available batch type statistical analysis systems.

(8) In short, the emphasis is on ease of use, rather than on statistical completeness.

In contrast, the emphasis in **SIPS** has been on making the system as useful and as powerful as possible for people at Oregon State University. The following attributes of the SIPS system result, at least in part, from this emphasis:

(1) Documentation for SIPS is adequate but not exhaustive. New users would probably require at least occasional telephone consultation with the SIPS staff.

(2) The system is highly optimized for the CDC Cyber computer system. It does not currently work on other computer systems, although such conversions may be done in the future.

(3) The Department of Statistics of Oregon State University maintains the SIPS system. User problems must be handled by this group only on a part-time basis. During the SIPS trial period at LRC, user problems were handled effectively.

(4) The interactive front-end (the user-interface) is a straightforward command processor with good error checking etc. As a result, the system is very efficient use.

(5) The output is generally presented in a clear and readable format, although there is not much flexibility in formatting tables and graphs.

(6) There is a large emphasis on providing as complete a set of statistical analyses as possible. New functions are continuously being added. However, there are no convenient links available to BMDP or SPSS.

(7) In short, the emphasis is on statistical completeness, rather than on universal useability.

The cost of the P-STAT system to LRC would be $5000 for the first year and $2000 for subsequent years; the cost of the SIPS system would be $5000 for the first year and nothing for subsequent years.

4.4 A Recommendation

Either the P-STAT 78 or the SIPS system would meet the minimum requirements of the STAT subsystem. However, it is felt that the superior statistical capabilities of the SIPS system, especially in the area of analysis of variance, far outweighs the I/O advantages of P-STAT for the intended usage in the PM Module. The expected superior efficiency of the SIPS system is another strong factor. For these reasons, it is recommended that LRC purchase the SIPS system for use as the STAT subsystem in the PM Module.

5. A Consistent File System for the PM Module

The Consistent File System links the DAT and STAT subsystems. This file system provides a means for passing data from the output of the experiments through the DAT and STAT subsystems. The primary task of the Consistent File System is to maintain the correspondence between the value of the data (e.g. the value of a particular performance measure) and the identities of the data (e.g. the name of the performance measure).

The design of the Consistent File System for the PM Module has two distinct parts: the design of **internal file structures** and the design of **external file structures.** Internal file structures means the choice of header content, record size, etc.: how does one identify or find a particular datum within a file. External file structures means the choice of file naming and accessing conventions: how does one identify and retrieve a particular file for analysis.

5.1 Internal File Structures

5.1.1 Design Objectives

The internal file structures of the Consistent File System are designed meet the following objectives:

(1) **Adaptable,** so that a wide variety of experiments and analyses may be accommodated. The system must handle experiments and performance measures which have not yet been conceived.

(2) **Self-documenting,** so that the files may be shared by users with little contact with each other. To achieve this, the system should store both the value and the **identity** (name, units, type, etc.) of each datum.

(3) **Expandable,** so that a subset of the users can make an addition to the file system with no impact on the other users, and without making existing files unreadable.

(4) **Upward compatible,** so that features which are not incorporated during the initial development can be added later.

(5) **Backward compatible,** so that existing file handling software (e.g. SIFT) can be used.

(6) **Convenient,** so that reading and writing the files requires just a few lines of code.

(7) **Efficient**, so that the computing resources, such as CPU time, disk space, memory, etc., are not unduly burdened.

5.1.2 SIFT Files

Over the past several years a file structure, or format, known as SIFT was developed at LRC. The goals of the SIFT file system were similar to the goals of the PM Consistent File System. It is not surprising then, that many, although not all, of the design objectives of the Consistent File System are met by SIFT. In this section, the SIFT file system is described, and its strengths and weaknesses indicated.

Figure 5.1 illustrates the format of a SIFT file. The file is divided into **blocks**, and each block is divided into **header** and **data** sections. The header section describes the layout or format of a record in the data section. The data section contains a set of data records in this format.

The header section consists of four records which specify the number of variables in each data record, and the name (e.g. input or error), units (e.g. grams, meters), a power of ten (e.g. 10**3), and a reference value for each variable.

The data section consists of any number of data records. Each record contains one value for each variable. The data section is terminated by a special END record.

The SIFT file system has the following strengths:

(1) **Self-documenting.** Each datum is identified by name and units.

(2) **Expandable.** Subsets of users may add variables to a particular file with little or no impact on other users. The limit of expandability is determined by the size of the read/write buffers.

(3) **Convenience.** It is extremely convenient for users to access SIFT files. Simple FORTRAN read and write statements are all that is required.

(4) **Efficiency.** Accessing SIFT files consumes little CPU time or program memory, since read and write requests are simple.

On the other hand, the SIFT file system appears to have the following weaknesses:

(1) **Adaptability.** A serious limitation on the adaptability of SIFT files is that they cannot handle arrays conveniently.

Block 1	Header Section	'NAME', N, FNAME(N) 'UNIT', N, FUNIT(N) 'ID', N, FDECML(N) 'ZERO', N, FREF(N)
	Data Section	'DATA', N, DATA(N) 'DATA', N, DATA(N) . . . . 'DATA', N, DATA(N) 'END', N, DATA(N)
Block 2	Header Section	'NAME', N, FNAME(N) 'UNIT', N, FUNIT(N) 'ID', N, FDECML(N) 'ZERO', N, FREF(N)
	Data Section	'DATA', N, DATA(N) 'DATA', N, DATA(N) . . . . 'DATA', N, DATA(N) 'END', N, DATA(N)
⋮	⋮	⋮

Figure 5.1 Format of a SIFT File

(There is no 5HDIMEN key, and N for data records would have to differ from N for all other records.) In addition, only real (floating point) numbers are currently handled, although some other (single-word) types could probably be added easily.

(2) **Efficiency.** SIFT files may make inefficient use of disk space since the record size is variable. The minimum size for a physical disk record is 64 words, so that if the record size were 16 (which is a reasonable size for scalar data) then 3/4 of the disk space used would be wasted. In addition, the user program must allocate a core buffer which is large enough to handle any record which it might encounter. Also, since these files are accessed sequentially, if some application requires random read access it will be very slow; random write access may be impossible. Finally, an application requiring very long records may be limited by the maximum physical record size allowed by the NOS operating system.

(3) **Upward compatibility.** Some features, such as vectors, could probably be implemented later in an upward compatible fashion. Other features, however, such as random access or uniform record length, are probably excluded.

5.1.3 SIFT Extended (SX) Files

In order to fulfill more of the design objectives of the PM Consistent File System, we have developed a file structure which is a natural extension of SIFT. This structure, which we call SIFT Extended or SX, combines the basic concepts of SIFT with a set of additional features.

As currently developed, the SX file structure may be thought of as being like the SIFT structure with a set of extensions. Many of these extensions are optional; they may be implemented or not, depending on the priorities of the LRC staff.

The following list of the SX extensions is divided into three groups. It is recommended that the first group be given a high priority for implementation. Many of the objectives of the PM Consistent File system would be met by implementing this group. It is recommended that the second group of extensions be given a medium priority. The extensions in this group are designed to improve the efficiency and increase the flexibility of the SX file system. It is recommended that the third group be given a low priority, and that their implementation be deferred until some experience with the SX file system has been obtained.

The following is a list of the high priority SX extensions to the SIFT file system. These extensions could be implemented quite simply, using FORTRAN READ and WRITE statements; no extra software would need to be written.

(1) **Arrays.** The SX files will support array data conveniently. There would be a header record which specified the size (i.e. dimension) of each variable in a data record.

(2) **Variable types.** The SX files will support varied data types conveniently. The basic types will be REAL, INTEGER, ad CHARACTER. The SX routines will not use this information, however; it will merely be storable and retrievable. Therefore, users will be able to add their own types, such as DATE, etc.

(3) **Expanded Header.** In the SIFT file system, the header contains a description of the format of the data records. In the SX file system, the header would be expanded so that it contained its own "data" record, called a **header record.** It would then contain a description of that record, along with a description of the data records. For example, the header record might serve to identify a run of an experiment (e.g. the date, subject, experimental conditions, etc.), while the data records would contain the results of the experiment (e.g. time series of input, error, etc.).

The following is a list of the medium priority SX extensions to the SIFT file system. Unlike the first set of extensions, these extensions would be implemented in a set of SX file utility routines. Therefore access to the SX files would not be via FORTRAN READ and WRITE statements, but rather via calls to these routines.

(4) **Buffered File Records.** To avoid the inefficiencies of very small disk file records, the problem of a maximum disk record size, and the requirement for a core buffer large enough for the largest record, SX file access will be buffered. Actual disk records will be a uniform size (probably 128 words), and are distinct from SX records which can be any size. Whenever an SX record is accessed, the required disk records will be automatically written in or out as required. A secondary advantage of buffered file records is that there is a considerable efficiency gain when the file records are small; many file records are read into memory with a single disk access.

(5) **Random Access.** The SX files will be accessible via random access. Readers will be able to access any existing record. In addition, random access will permit header information, such as the location of the last record or the time the file was last written into, to be updated as needed. The current

NOS FORTRAN supports random access files (OPENMS, READMS and WRITMS). However, it imposes the requirement that an array be kept in the users memory space (FL) of length equal to one plus the maximum number of disk records in the file. This could be a significant problem if several large SX files must be open concurrently. It is expected that a FORTRAN 77 conforming compiler will become available within a year or so; such a compiler should support random access without this overhead.

(6) **Read/Write Conventions.** Data may be written into an SX file as an entire record or as a set of fields within a record. Data is read from an SX file **only** as a set of fields within a record. This restriction on readers is intended to guarantee the expandability of the SX file structure. For example, if a reader assumed that the position of the fields within a record was not going to change, he might read whole records and extract the fields himself. The resulting code, however, would be vulnerable to a change in the structure of a record, such as the addition of a new field. This extension would, however, impose an inefficiency, hopefully small, on readers.

The following three additional extensions are given low priority; their implementation should probably be deferred until some experience with the SX file system has been obtained. However, care should be taken in the design not to preclude incorporating these extensions at a later time.

(7) **SX Supplied Fields.** The SX file system might automatically supply one or two fields a each record is created. Two possibilities are Record-Number and Entry-Time.

(8) **Access Control.** The SX file system should probably contain some kind of access control. This control would have to work within the constraints imposed by the NOS operating system. A minimum form of access control would require users to specify a mode of access (e.g. read, append, or write) when they opened an SX file. The SX routines would then enforce access mode restrictions.

(8) **SX Directory.** For many applications where SX files contained multiple Header/Data blocks, it would be useful to have a directory of the blocks. This directory would be written before the first block.

5.1.4 SX File Structure

Figure 5.2 illustrates the format of one block of an SX file. As in SIFT files, the header section describes the layout or format of

```
==========================================================
|           | Pre-Header:                                |
|           |      LPH, NREC, DATECR, TIMECR, USERCR     |
|           ----------------------------------------------
|           | Header Record Descriptor:                  |
|           |      NFLDH, FNAMEH(NFLDH), FSIZEH(NFLDH),  |
|  Header   |             FUNITH(NFLDH), FTYPEH(NFLDH)   |
|  Section  ----------------------------------------------
|           | Header Record:                             |
|           |      Field 1, Field 2, ... , Field NFLDH   |
|           ----------------------------------------------
|           | Data Record Descriptor:                    |
|           |      NFLD, FNAME(NFLD), FSIZE(NFLD),       |
|           |            FUNIT(NFLD), FTYPE(NFLD)        |
----------------------------------------------------------
|           | Data Record 1:                             |
|           |      Field 1, Field 2, ... , Field NFLD    |
|           ----------------------------------------------
|           | Data Record 2:                             |
|           |      Field 1, Field 2, ... , Field NFLD    |
|  Data     ----------------------------------------------
|  Section  |        .        .              .           |
|           |        .        .              .           |
|           |        .        .              .           |
|           ----------------------------------------------
|           | Data Record NREC:                          |
|           |      Field 1, Field 2, ... , Field NFLD    |
==========================================================
```

Figure 5.2 Format of One Block of an SX File

a record in the data section, and the data section contains a set of records in this format. The major difference between the SX and SIFT files blocks is that the SX header contains additional information.

The Header section contains four records: a **pre-header**, a **header record descriptor**, a **header record**, and a **data record descriptor**. The Data section contains **data records**. All of these records in both the Header and data sections are divided into **fields**. The fields of the pre-header, header record descriptor, and data record descriptor are fixed. The fields of the header record and data records are determined by the users. These fields are all described below and are summarized in Table 5.1.

The pre-header contains the length of the pre-header (LPH), and the number of records in the data section (NREC). It might also contain the date and time the block was created (DATECR, TIMECR) as well as the user who created the block (USERCR). The header record descriptor contains the number of fields in the header record (**NFLDH**), and the name, size units and type of each field (**FNAMEH**(NFLDH), **FSIZEH**(NFLDH), **FUNITH**(NFLDH), **FTYPEH**(NFLDH)). The header record contains NFLDH fields as described by the header record descriptor. The data record descriptor is analogous to the header record descriptor and contains the number of fields in each data record (**NFLD**), and the name, size, units and type of each field (**FNAME**(NFLD), **FSIZE**(NFLD), **FUNIT**(NFLD), **FTYPE**(NFLD)). The data records contain NFLD fields as described by the data record descriptor.

Figure 5.3 illustrates the overall format of an SX file with multiple blocks. The first block, called the **SX Header** contains the location of the SX Directory (**LOCDIR**), the location of the current EOF (**LOCEOF**), the current number of blocks (**NBLK**), and the maximum number of allowable blocks (**MAXBLK**). The **SX Directory** contains the location of the start of each block in the SX file.

5.1.5 SX File Access

In this section, the means of accessing SX files is described. As noted in Section 5.1.3, if only the first three high priority SX extensions to the SIFT file system are implemented, then the SX files could be accessed using just FORTRAN READ and WRITE statements; no extra software would need to be written. If however, additional extensions are desired, then the following set of SX utility subroutines would provide users with a convenient mechanism for accessing SX files. The following list of routines would allow users to open and close SX files, to read and write SX record descriptors, and to read and write SX records.

(1) Opening SX Files.

```
==================================================
| SX Header       | LOCDIR, LOCEOF, NBLK, MAXBLK |
==================================================
|                 | Location of Block 1          |
|                 | Location of Block 2          |
|                 |    .     .    .   .          |
|                 |    .     .    .   .          |
| SX Directory    |    .     .    .   .          |
|                 | Location of Block NBLK       |
|                 |    .     .    .   .          |
|                 |    .     .    .   .          |
|                 |    .     .    .   .          |
|                 | Location of Block MAXBLK     |
==================================================
|                 | Header Section               |
| Block 1         --------------------------------
|                 | Data Section                 |
==================================================
|                 | Header Section               |
| Block 2         --------------------------------
|                 | Data Section                 |
==================================================
.                 .                              .
.                 .                              .
.                 .                              .
==================================================
|                 | Header Section               |
| Block NBLK      --------------------------------
|                 | Data Section                 |
==================================================
```

Figure 5.3 Overall Format of an SX File with Multiple Blocks

SXNEW: Opens a new SX file
SXOLD: Opens an existing SX file

These two routines require that the files to be opened be specified by **name** at run-time. The current NOS FORTRAN compiler does not support this capability. A set of assembly language (COMPASS) routines with this capability, however, has been developed at LRC. In addition, a FORTRAN 77 conforming compiler, which should become available within about a year, would also support this capability.

(2) Writing SX Record Descriptors.

SXDEFH: Writes a header record descriptor
SXDEF: Writes a data record descriptor

(3) Reading SX Record Descriptors.

SXRECH: Returns information about a set of fields of a header record, specified by field numbers.
SXFLDH: Returns information about a set of fields of a header record, specified by field names.
SXREC: Returns information about a set of fields of a data record, specified by field numbers.
SXFLD: Returns information about a set of fields of a data record, specified by field names.

(4) Writing SX Records.

SXWRRH: Writes an entire header record.
SXWRFH: Writes a set of fields in a header record, specified by field numbers.
SXWRR: Writes an entire data record.
SXWRF: Writes a set of fields in a data record, specified by field numbers.

(5) Reading SX Records.

SXRDFH: Reads a set of fields from the header record, specified by field numbers.
SXRDF: Reads a set of fields from a data record, specified by field numbers.

Note that header and data record descriptors are written as entire records, whereas they may be read by field names or field numbers. This is because while writers would presumably know what fields are to be written, readers might not know any of the field names or might desire to search for a specific field name. Also, writers can access header and data records by either entire records or by field numbers, whereas readers must access these records by field numbers (having obtained these field numbers by reading header or data record descriptors).

5.2 External File Structures

5.2.1 Design Objectives

The external file structures of the Consistent File System simplify the task of identifying and retrieving a particular file for analysis. The system is in fact a file-naming convention, supported by appropriate software, with multi-field file names. The fields would serve to identify the following attributes of the file:

(1) The **user** who "owns" the file: his group and his name.

(2) The **experiment** to which the file pertains: the name of the experiment, the pilot and run number.

(3) The **type** of file: time-series data, processed data, etc.

(4) The **version** number of the file.

These file naming conventions would probably also be very useful for keeping track of programs pertaining to the PM Module, such as FORTRAN source files, documentation files, etc.

5.2.2 Fields of File Names

File names are divided into four fields, **User**, **Experiment**, **Type**, and **Version**, corresponding to the above four attributes of a file. The fields could be separated by a character such as "/":

```
User/Experiment/Type/Version
```

To accomodate files of other types, i.e. not experimental data, the Experiment field could be given the more general name of **FileName**:

```
User/FileName/Type/Version
```

These fields could, in turn, be further divided into subfields, to more specifically identify the file. The sub-fields could be separated by a character such as ".". Thus a file name with four fields could have the following sub-fields:

```
User     ==> Group.UserName

FileName ==> Experiment.Pilot.Run
```

```
Type      ==> MajorType.MinorType

Version   ==> MajorVersion.MinorVersion
```

5.2.3 Implementing External File Structures

Implementing these external file structures, i.e. long, multi-field file names, directly under the NOS operating system would be extremely difficult, if not impossible. The NOS operating system is limited to file names of seven letters or less, with each user having his own set of file names. A feasable approach, however, would be to write a program which somehow maintained a correspondence between long user-oriented file names, and short machine-oriented file names. At the present time, two programs are in existence at LRC, which approach this capability: The File Information and Cataloging System (**FICS**), and The Permanent File Cataloging System (**CCATSYS**). The applicability of these two programs to the needs of the PM external file structures is discussed below.

5.2.3.1 File Information and Cataloging System (FICS)

FICS provides a method for identifying, cataloging and displaying contents of the permanent file system. The system is built as an inverted tree structure, composed of information nodes. The top level node is called the root node. Below the root node are other nodes, connected to each other by branches. Some of the nodes, called data base nodes, are associated with file entries which describe actual NOS files. Each data base node may have file entries for a number of NOS files.

One way of using FICS would be to associate the FICS node levels with the User field and its sub-fields Group, UserName, or perhaps some others. The Type field would be handled by the "Type Description" portion of the file entry in the data base. The remaining fields, FileName and Version, would have to be handled by the 80 character File Description provided by FICS.

Another way of creating the above external file structures using FICS would be to associate each node level with a field (or subfield), and each node at that level with an instance of that field (or subfield). The file entry could then be used to complete the file name and to store other information about the file. A disadvantage of this approach is that only a priviledged user, called the Tree Boss, is allowed to create new nodes. Therefore, only a Tree Boss can name files.

There are three disadvantages of FICS. First, the system appears to be somewhat cumbersome. Although practice users might become proficient, it seems that for this application FICS requires an unduly large time investment to learn to use. Second, the procedure for adding a new file to FICS, while not terribly inconvenient, is manual and requires some effort on the part of the user, both to remember to perform the operation as well as to do it. Third, although FICS provides a means of associating seven letter file names with longer, more meaningful names, users are still required to invent unique seven letter file names for all their files. A possible extension to FICS would be for the system to generate unique file names which the user might never even see directly.

5.2.3.2 Permanent File Cataloging System (CCATSYS)

CCATSYS provides another system for identifying, cataloging and displaying contents of the permanent file system. CCATSYS is somewhat smaller than FICS, and is analogous to a single data base node of FICS. Consequently, it is suitable for each user to maintain his own CCATSYS catalog, obviating the User field. In a manner similar to FICS, the Type field would be handled by the File Type facility of CCATSYS, and the remaining fields FileName and Version would be handled by the 37 character File Description provided by CCATSYS.

CCATSYS has one major advantage over FICS, although it shares some of the disadvantages of FICS. The advantage is that CCATSYS appears to be extremely easy to use, much easier than FICS. On the other hand, as with FICS, the procedure for adding a new file to CCATSYS is manual, and users are still required to generate unique seven letter file names. The major disadvantage of CCATSYS is that only 37 character File Descriptions are permitted versus the 80 character descriptions permitted by FICS. In addition, whereas a FICS "tree" could serve a large number of users, individual CCATSYS files are required for each user. On the other hand, CCATSYS users could easily read each others CCATSYS files and printouts.

Overall, it appears that CCATSYS would be the cataloging system of choice for implementing the PM External File Structures. The advantage of the relative convenience of CCATSYS far outweighs the disadvantage of not being able to share the cataloging system across users.

5.3 Examples of Internal and External File Structures

The following examples illustrate how these internal and external file structures could be used. Suppose Burnell McKissick of the simulation group at LRC is running an experiment called "G-SEAT", using four pilots (ABC, DEF, GHI, and JKL). See Ashworth, et al. (1977) for the details of this study.

5.3.1 Single-Run Data Files

Each run of the experiment would produce a Single-Run data file with a name such as:

```
LRC-SIM.MCKISSICK/G-SEAT.DEF.4/SX.DATA/1.0
```

which would correspond to subject DEF and run 4. If the run had to be redone for some reason, the new file might be version 1.1. The header record of this file would include the following fields which identify the file, and possibly some others which more specifically identify the experimental conditions:

NFLDH = 12

FNAMEH	**FSIZEH**	**FUNITH**	**FTYPEH**	**VALUE**
Group	1	-	CHAR	LRC-SIM
UserName	1	-	CHAR	MCKISSICK
Experiment	1	-	CHAR	G-SEAT
Pilot	1	-	CHAR	DEF
Run	1	-	CHAR	4
MajorType	1	-	CHAR	SX
MinorType	1	-	CHAR	DATA
MajorVsn	1	-	INTEGER	1
MinorVsn	1	-	INTEGER	0
Condition	1	-	CHAR	SEAT-ON
SampleRate	1	SAMP/SEC	REAL	16.0
Date	1	-	DATE	11-FEB-79

The data records of this file would consist of eleven fields, one for each system state:

NFLD = 11

FNAME	**FSIZE**	**FUNIT**	**FTYPE**
TKE	1	DEGREES	REAL
TKL	1	DEGREES	REAL
TKC	1	DEGREES	REAL
.	.	.	.
.	.	.	.
.	.	.	.
T	1	SAMPLES	INTEGER

5.3.2 Single-Run Measurement Files

One analysis scheme would be to analyze each such Single-Run data file to produce a corresponding Single-Run measurement file:

LRC-SIM.MCKISSICK/G-SEAT.DEF.4/SX.MEAS/1.0

The header of this file would include the same fields as the preceding data file except that the file type would be SX.MEAS instead of SX.DATA.

NFLDH = 12

FNAMEH	FSIZEH	FUNITH	FTYPEH	VALUE
Group	1	-	CHAR	LRC-SIM
UserName	1	-	CHAR	MCKISSICK
Experiment	1	-	CHAR	G-SEAT
Pilot	1	-	CHAR	DEF
Run	1	-	CHAR	4
MajorType	1	-	CHAR	SX
MinorType	1	-	CHAR	MEAS
MajorVsn	1	-	INTEGER	1
MinorVsn	1	-	INTEGER	0
Condition	1	-	CHAR	SEAT-ON
SampleRate	1	SAMP/SEC	REAL	16.0
Date	1	-	DATE	11-FEB-79

The file would contain just a single data record with one field for each of the 90 performance measures:

NFLD = 90

FNAME	FSIZE	FUNIT	FTYPE
MTKE	1	DEGREES	REAL
MTKL	1	DEGREES	REAL
MTKC	1	DEGREES	REAL
.	.	.	.
.	.	.	.
.	.	.	.
TS+	1	SECONDS	REAL
TS-	1	SECONDS	REAL

5.3.3 Pilot-Summary Measurement Files

Later, one could combine these single-run measurement files across runs to produce pilot-summary measurement files:

```
LRC-SIM.MCKISSICK/G-SEAT.DEF.ALL/SX.MEAS/1.0
```

where ALL indicates all runs. The header record of this file would consist of the following fields:

NFLDH = 10

FNAMEH	FSIZEH	FUNITH	FTYPEH	VALUE
Group	1	-	CHAR	LRC-SIM
UserName	1	-	CHAR	MCKISSICK
Experiment	1	-	CHAR	G-SEAT
Pilot	1	-	CHAR	DEF
Run	1	-	CHAR	ALL
MajorType	1	-	CHAR	SX
MinorType	1	-	CHAR	DATA
MajorVsn	1	-	INTEGER	1
MinorVsn	1	-	INTEGER	1
SampleRate	1	SAMP/SEC	REAL	16.0

The data records of this file would consist of three fields identifying the run, plus fields for the 90 performance measures:

NFLD = 93

FNAME	FSIZE	FUNIT	FTYPE
Run	1	-	CHAR
Condition	1	-	CHAR
Date	1	-	DATE
MTKE	1	DEGREES	REAL
MTKL	1	DEGREES	REAL
MTKC	1	DEGREES	REAL
.	.	.	.
.	.	.	.
.	.	.	.
TS+	1	SECONDS	REAL
TS-	1	SECONDS	REAL

5.3.4 Overall-Summary Measurement Files

Still later, one could combine the Pilot-Summary measurement files to produce an Overall-Summary measurement file:

LRC-SIM.MCKISSICK/G-SEAT.ALL.ALL/SX.MEAS/1.0

where ALL.ALL represents all pilots and all runs. The header record of this file would include the following fields:

NFLDH = 10

FNAMEH	**FSIZEH**	**FUNITH**	**FTYPEH**	**VALUE**
Group	1	-	CHAR	LRC-SIM
UserName	1	-	CHAR	MCKISSICK
Experiment	1	-	CHAR	G-SEAT
Pilot	1	-	CHAR	ALL
Run	1	-	CHAR	ALL
MajorType	1	-	CHAR	SX
MinorType	1	-	CHAR	DATA
MajorVsn	1	-	INTEGER	1
MinorVsn	1	-	INTEGER	1
SampleRate	1	SAMP/SEC	REAL	16.0

The data records of this file would consist of four fields identifying the pilot and the run, plus fields for the 90 performance measures:

NFLD = 94

FNAME	**FSIZE**	**FUNIT**	**FTYPE**
Pilot	1	-	CHAR
Run	1	-	CHAR
Condition	1	-	CHAR
Date	1	-	DATE
MTKE	1	DEGREES	REAL
MTKL	1	DEGREES	REAL
MTKC	1	DEGREES	REAL
.	.	.	.
.	.	.	.
.	.	.	.
TS+	1	SECONDS	REAL
TS-	1	SECONDS	REAL

Finally, this Overall-Summary measurement file would be in a form suitable for statistical analysis by the STAT subsystem. In fact, one could obtain preliminary statistical results from this experiment by analyzing Pilot-Summary files or even Single-Run files with the STAT subsystem.

6. References

Anderson, R.E. and F.M. Sim (1977). "Data management and statistical analysis in social science computing," **American Behavioral Scientist, 20:** 367-409.

Ashworth, B.R., B.T. McKissick and D.J. Martin, Jr. (1977). "Objective and subjective evaluation of the effects of a G-seat on pilot/simulator performance during a tracking task," Presented at the Tenth NTEC/Industry Conference, Orlando, FL:

Barr, A.J., et al. (1976). **A User´s Guide to SAS 76**, Raleigh, NC: SAS Institute Inc.

Brainerd, W., Ed. (1978). "Fortran 77," **Communications of the ACM,** 21, 806-820.

Buhler, R. and S. Buhler (1979). **P-STAT 78 User´s Manual**, Princeton, NJ: P-STAT Inc.

CDC (1979). **Fortran Extended Version 4 Reference Manual**, Sunnyvale, CA: Control Data Corporation.

CDC (1979). **NOS Version 1 Reference Manual**, Sunnyvale, CA: Control Data Corporation.

Coover, E.R., et al. (1974). "Design of an optimally compatible social data analysis system: the first steps," **Social Science Information,** 13: 105-146.

Donaldson, J. and B. Ankeney (1978). **File Information and Cataloging System (FICS)**, Computer Sciences Corporation.

Nie, N.H. and C.H. Hull (1979). **The SCSS Conversational System Release 3.1 Preliminary User´s Manual**, Chicago, IL: SPSS Inc.

Rowe, K. and J.A. Barnes (1978). **Statistical Interactive Programming System (SIPS) Beginner´s Manual**, Corvallis, OR: Department of Statistics, Oregon State University.

Rowe, K. and J.A. Barnes (1978). **Statistical Interactive Programming System (SIPS) Command Reference Manual**, Corvallis, OR: Department of Statistics, Oregon State University.

Russell, C., et al. (1979). **RS/1 User´s Manual**, Cambridge, MA: Bolt Beranek and Newman Inc.

Shucany, W.R., P.D. Minton and B.S. Shannon, Jr. (1972). "A survey of statistical packages," **Computing Surveys**, 4, 65-79.

Whitehead, S. (1979). **RS/1 Primer**, Cambridge, MA: Bolt Beranek and Newman Inc.

Howell, R. D., Guillebeau, W. M., Long, R. L. (1980). "Interface File Tape Concept", SDC Integrated Services, Inc., NASA CR 159284.

End of Document